INRIA

CENTRE DE RENNES

IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 574

LE DSPA
UN PIPELINE SYNCHRONISÉ PAR LES DONNÉES

Yvon JEGOU

Octobre 1986

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

le DSPA
un pipeline synchronisé par les données

the DSPA
a data synchronized pipeline

Yvon JEGOU

Publication Interne n° 311
Septembre 1986
70 pages

Résumé

Le domaine d'application d'un multiprocesseur est d'autant plus large que ses processeurs élémentaires sont capables d'exécuter rapidement du code scalaire. Pour permettre une utilisation optimale de ses divers composants, un processeur est nécessairement pipeline, structuration qui entraîne des contraintes très sévères sur le séquencement des instructions scalaires.

L'organisation du processeur DSPA permet le séquencement et le décodage de plusieurs instructions simultanément. Elle permet aussi la gestion d'unités fonctionnelles dont le comportement ne peut être prédit ni à la compilation, ni même au décodage des instructions.

Des performances élevées peuvent être atteintes sur un processeur DSPA sans faire appel à des logiciels complexes d'optimisation comme les vectoriseurs.

Abstract

The application field of a multiprocessor depends heavily on the ability of its processing elements to run scalar codes at high speed. But, in order to get optimal usage of their componants, these processors are necessarily pipelined. This structure introduces very strong constraints on scalar instruction sequencing.

The organization of the DSPA allows sequencing and decoding of many instructions in parallel. Moreover, it allows optimal management of pipelined funtional units, even when their behavior cannot be foreseen neither at compile time nor at decode time.

High performances can be obtained without the need of complex optimizing softwares as vectorizers on a DSPA processor.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L.A. 227)
UNIVERSITÉ DE RENNES 1 I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Introduction

Généralités sur le pipeline

Le comportement général d'un multiprocesseur fortement couplé dépend principalement de la capacité de ses processeurs élémentaires à exécuter rapidement du code scalaire pour deux raisons.

1- une partie plus ou moins importante de tout algorithme est nécessairement traduit en code scalaire. Le temps d'exécution de cette portion scalaire ne dépend pas du nombre de processeurs et peut dans certains cas dominer le temps total d'exécution.

2- Pour un algorithme donné, une augmentation du nombre des processeurs se traduit nécessairement par une diminution proportionnelle de la longueur des vecteurs traités par chaque processeur.

Le domaine d'application d'un tel multiprocesseur est donc d'autant plus large que ses processeurs sont capables d'exécuter rapidement du code scalaire. Contrairement à ce qui se passe dans le cas de codes vectoriels, les performances d'un processeur scalaire sont surtout liées au débit de séquencement des instructions sur le processeur. Par exemple, pour obtenir des performances honnêtes sur des instructions de type CRAY 1 par rapport aux performances obtenues sur des instructions vectorielles, il faudrait séquencer et décoder en moyenne 3 ou 4 instructions par cycle mémoire du processeur. De plus, pour optimiser l'utilisation des divers composants de l'architecture (mémoires entrelacées, opérateurs segmentés etc.), un tel processeur est nécessairement pipeline.

Cette structuration pipeline entraîne des contraintes particulières sur le séquencement des instructions. La première contrainte provient du fait que le résultat produit par une instruction décodée pendant un cycle donné n'est généralement pas disponible au cycle suivant mais un certain nombre de cycles (latence) plus tard. La connaissance des latences des opérateurs permet de réordonner des séquences linéaires de code dans de nombreux cas, à la compilation. Mais ce réordonnement est très délicat, voire impossible, dès qu'il y a des instructions de rupture de séquence. Il

n'est en général pas possible d'optimiser, à la compilation, l'exécution d'une boucle courte (contenant un nombre trop faible d'instructions). De plus, les latences de toutes les instructions ne sont pas toujours connues statiquement, ni même au moment du décodage. C'est le cas sur les multiprocesseurs où certaines ressources sont partagées (mémoire).

Une deuxième contrainte très sévère provient du comportement de la mémoire dans une architecture pipeline. Il suffit de constater qu'une instruction d'un langage de haut niveau commence généralement par une lecture mémoire et se termine généralement par une écriture dans cette même mémoire pour vérifier qu'une traduction instruction par instruction par un compilateur entraîne un vidage systématique de tous les pipelines entre chaque instruction. Là aussi, après analyse de dépendance sur les instructions d'accès à la mémoire, il est possible d'entrelacer les séquences de codes issues de la traductions d'instructions successives, et donc de conserver une certaine activité dans les pipelines. Mais cette analyse de dépendance échoue dès que les données accédées ne sont plus des scalaires ou des vecteurs rangés régulièrement dans la mémoire. En fait, dès que l'analyse des dépendances n'autorise pas la vectorisation d'une boucle, elle interdit également l'optimisation du code séquentiel. Cet entrelacement échoue également en présence d'instructions de contrôle, et donc dans le cas de boucles courtes.

Le comportement d'un pipeline ne peut donc pas être entièrement optimisé par le logiciel. L'entrelacement des instructions de lecture et d'écriture en mémoire doit être décidé à l'exécution après analyse des dépendances sur les adresses effectives. Cet entrelacement se traduit par un doublement contrôlé des instructions d'écriture par des instructions de lecture.

Le processeur DSPA que nous introduisons dans ce document répond aux objectifs suivants :

- séquencement et décodage de plusieurs instructions en parallèle ;

- doublement des écritures par les lectures sur la mémoire, doublement contrôlé dynamiquement à l'exécution ;

- utilisation optimale d'opérateurs pipelines et de mémoires entrelacées dans un contexte multiprocesseurs ;

- production de code peu complexe.

L'intégration de ce processeur dans une architecture multiprocesseur est définie en [Jégou 86a].

le pipeline synchronisé par les données - DSPA

Un processeur DSPA est composé d'un ensemble d'unités fonctionnelles spécialisées comme la mémoire, l'additionneur flottant, le multiplieur flottant, l'unité de calcul sur les entiers etc. Pour permettre un séquencement rapide des instructions d'un programme, il faut réduire au minimum la complexité des opérations de décodage et de contrôle à réaliser pendant ce séquencement. Le principe de base du séquencement d'un DSPA est très simple : il consiste à émettre chaque instruction vers l'unité fonctionnelle qui doit l'exécuter. Les opérations à dérouler se limitent donc au décodage du numéro d'unité fonctionnelle destinatrice, à l'incrémentation du compteur ordinal et à l'extraction de l'instruction suivante. Ce séquencement implicite s'arrête dès que l'unité fonctionnelle destinatrice est le séquenceur, auquel cas, le séquencement est repris à l'adresse délivrée par ce séquenceur. Une instruction d'un programme de DSPA est donc décomposée en deux champs

UF | opération

Le champs opération ne concerne que l'unité fonctionnelle désignée et n'est pas analysé pendant la phase de séquencement.

Pour qu'un tel système fonctionne correctement, les conditions suivantes doivent être respectées :

● Une instruction ne génère d'activité que sur l'unité fonctionnelle qui l'exécute. Cette condition est nécessaire pour éviter toute analyse du champs opération de l'instruction pendant la phase de séquencement.

● Chaque unité fonctionnelle possède une mémoire d'instructions, condition nécessaire pour ne pas lier le séquencement à la disponibilité des unités fonctionnelles.

● L'échange des données entre les unités fonctionnelles doit être défini de manière à éviter l'introduction de mécanismes globaux et donc de décoder le champs opération des instructions pendant la phase de séquencement.

● L'association des données aux instructions doit également être traitée localement aux unités fonctionnelles.

1 Le modèle de base

Dans le modèle de base, chaque unité fonctionnelle possède une entrée d'instructions, une ou plusieurs entrées logiques et une sortie de données. Chaque entrée logique d'unité fonctionnelle peut être connectée directement aux sorties logiques de toutes les unités fonctionnelles. En général, une instruction a la forme suivante :

UF | opération | paramètres | sources | destination

En fait, comme le décodage d'une instruction se fait localement à l'unité fonctionnelle concernée, le formatage est libre. Le champs opération définit l'ordre à exécuter. Le champs paramètre contient des informations complémentaires comme un numéro de registre local etc. Un champ source correspond à chaque entrée logique et désigne l'unité fonctionnelle productrice de l'opérande correspondant, et donc une connexion sortie → entrée logique. Le champs destination désigne une entrée logique d'unité fonctionnelle vers laquelle le résultat doit être émis, et donc une connexion.

Chaque unité fonctionnelle possède un séquenceur et un décodeur d'instructions. Sur une unité fonctionnelle, une instruction est exécutée lorsque ses opérandes sont présents, le résultat étant émis vers le destinataire lorsqu'il est calculé. Pour permettre à la fois une mise en relation simple de chaque instruction avec ses opérandes et un certain asynchronisme du fonctionnement des unités fonctionnelles, chaque point de

connexion est une file d'attente FIFO, la mémoire d'instruction de chaque unité fonctionnelle étant également gérée en FIFO. L'unicité d'interprétation d'une séquence d'instruction d'un DSPA est garantie dès lors que les conditions suivantes sont respectées :

1↦ chaque unité fonctionnelle exécute ses instructions propres dans l'ordre où elles ont été extraites du programme (ordre de séquencement global).

2↦ chaque unité fonctionnelle émet ses résultats dans l'ordre d'exécution de ses instructions.

3↦ les données sont extraites des points de connexion dans l'ordre où elles y ont été déposées (gestion strictement FIFO des files d'attente).

Les condition 1 et 3 sont garanties par l'utilisation de FIFO dans les points de connexions et les mémoires d'instructions. La condition 2 doit être respectée dans le séquencement interne de chaque unité fonctionnelle. Ces trois conditions peuvent sembler très strictes. Elles peuvent être assouplies. Par exemple, la condition 2 n'est strictement nécessaire que sur des instructions possédant le même destinataire (c'est un ordre partiel). Deux instructions d'une même unité fonctionnelle qui ne font référence ni aux mêmes sources ni au même destinataire sont indépendantes et peuvent donc être exécutées dans un ordre quelconque. Cependant, nous cherchons à montrer par la suite que ce modèle de pipeline a une très bonne efficacité lorsque les trois conditions précédentes sont respectées. La figure 1 montre un exemple de processeur DSPA dans lequel les bus horizontaux transportent les résultats produits par chaque unité fontionnelle, les bus verticaux correspondent aux entrées logiques, les points de connexion correspondent aux FIFOs d'échange des données, et une FIFO d'instruction est associée à chaque unité fonctionnelle. La figure 2 montre quatre FIFOs localisées aux croisements de deux sorties logiques et de deux entrées logiques. Certaines connexions comme, par exemple, entre une sortie d'opérateur flottant et l'entrée d'adresses de la mémoire présentent peu d'intérêt et peuvent être supprimées.

Les limites pratiques au débit de séquencement des instructions d'un DSPA sont :

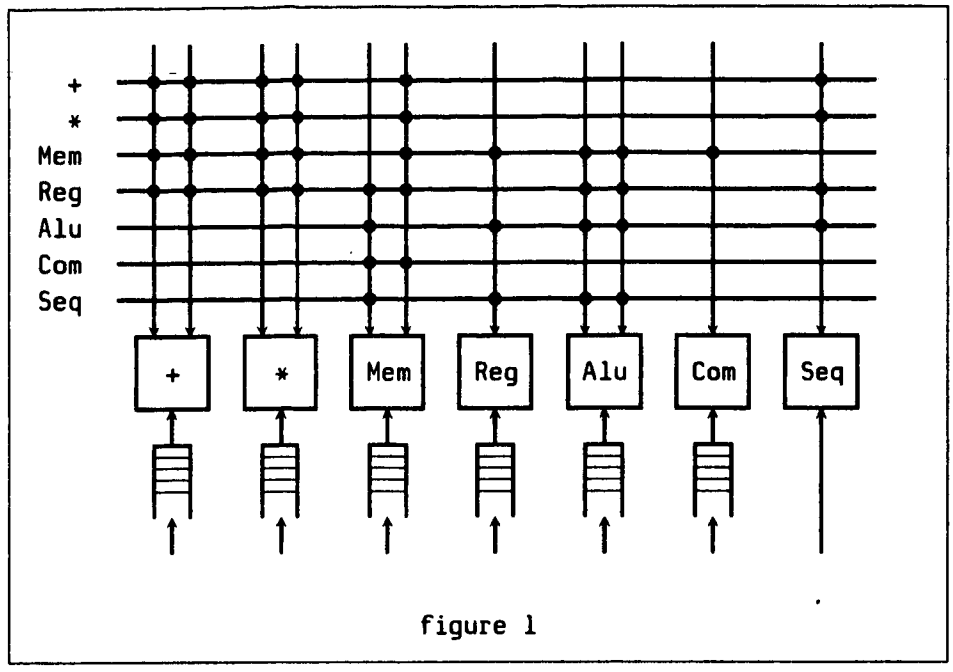

figure 1

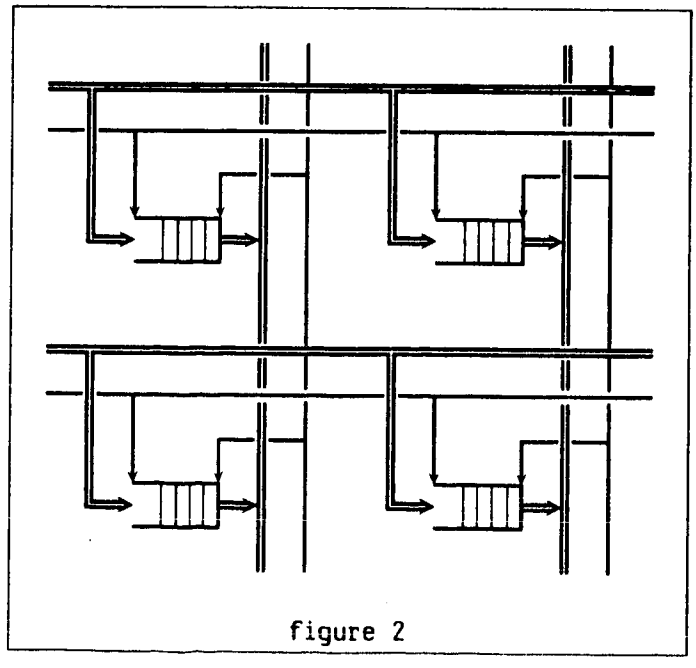

figure 2

- le débit de la mémoire d'instructions du processeur.

- le temps de décodage du champs UF d'une instruction.

- le débit en écriture dans les FIFOs d'instructions des unités fonctionnelles.

- la vitesse d'exécution des instructions du séquenceur.

Ce débit peut être très élevé. En effet, plusieurs instructions peuvent être séquencées en parallèle. Une première solution consiste à rassembler une instruction par unité fonctionnelle du DSPA dans chaque instruction d'un programme comme on le fait généralement pour les micro-programmes. Mais les débits demandés aux unités fonctionnelles étant généralement différents, des solutions intermédiaires plus compactes comme le partage d'un champs par deux unités fonctionnelles peuvent être adoptées. Ces partages sont rendu possibles par l'asynchronisme dans le fonctionnement des UFs et par la présence de mémoires locales d'instructions. Sur une architecture équilibrée à une opération flottante par accès mémoire, un champs peut être associé à l'UF mémoire, un autre champs étant partagé par l'additionneur et le multiplieur flottant, un troisième champs étant alloué à l'unité de calcul des adresses etc... Une autre possibilité plus souple, mais un peu plus complexe, consiste à ne coder qu'une seule instruction par mot de la mémoire de programme et à les extraire par paquets de mots contigüs mais il faut alors traiter le cas où plusieurs instructions d'un même paquet sont destinées à la même unité fonctionnelle. Dans tous les cas de séquencement parallèle, la signification des instructions à destination du séquenceur doit être clairement définie. La solution la plus générale consiste à considérer que l'ordre au séquenceur termine le paquet d'instructions. Une solution simplifiée consiste à regrouper les instructions par paquets de taille fixe à la génération du code. Une instruction du séquenceur n'a alors aucun effet sur les instructions du paquet d'instructions où elle se trouve. Les adresses de saut doivent alors être des adresses de paquets. On peut dans ce cas limiter à un le nombre d'instructions à destination d'une même unité fonctionnelle dans un même paquet, ce qui permet de simplifier la gestion des mémoires d'instructions des unités fonctionnelles.

Voici quelques exemples d'unités fonctionnelles typiques.

2 l'additionneur flottant

Une instruction de l'additionneur flottant a le format général suivant :

+f | op | gauche | droite | destinataire

Les champs gauche et droite désignent les unités fonctionnelles émettrices des données de l'opération et correspondent aux entrées logiques gauche et droite de l'opérateur. L'un de ces champs est absent pour les opérations unaires. Le champs destinataire désigne une entrée logique d'unité fonctionnelle vers laquelle le résultat doit être émis. Par exemple, l'instruction suivante d'un programme

+f | + | Mémoire | *f | dMémoire

est rangée dans la FIFO d'instruction de l'additionneur flottant +f pendant le séquencement. Son exécution par l'additionneur provoque l'exécution de l'opération + sur une donnée en provenance de l'UF Mémoire (émise par l'unité mémoire vers l'entrée logique gauche +fg de l'additionneur) et une donnée en provenance de l'UF multiplieur *f (émise par le multiplieur flottant vers l'entrée logique droite +fd de l'additionneur), le résultat étant émis vers l'entrée logique données dMémoire de la mémoire dans la FIFO associée à la connexion +f → dMémoire. La figure 3 donne un exemple d'intégration d'opérateur dans un processeur DSPA. Les entrées de l'opérateur communiquent directement avec les FIFOs sources. Le bus de sortie de l'opérateur émet directement vers les FIFOs de destination. Aucune contrainte n'interdit l'émission d'un résultat vers l'une des entrées propres de l'opérateur (rebouclage direct). Un séquenceur et un décodeur d'instructions sont associés à l'opérateur. Son fonctionnement est très simple, l'exécution d'une instruction étant décomposée en cinq phases :

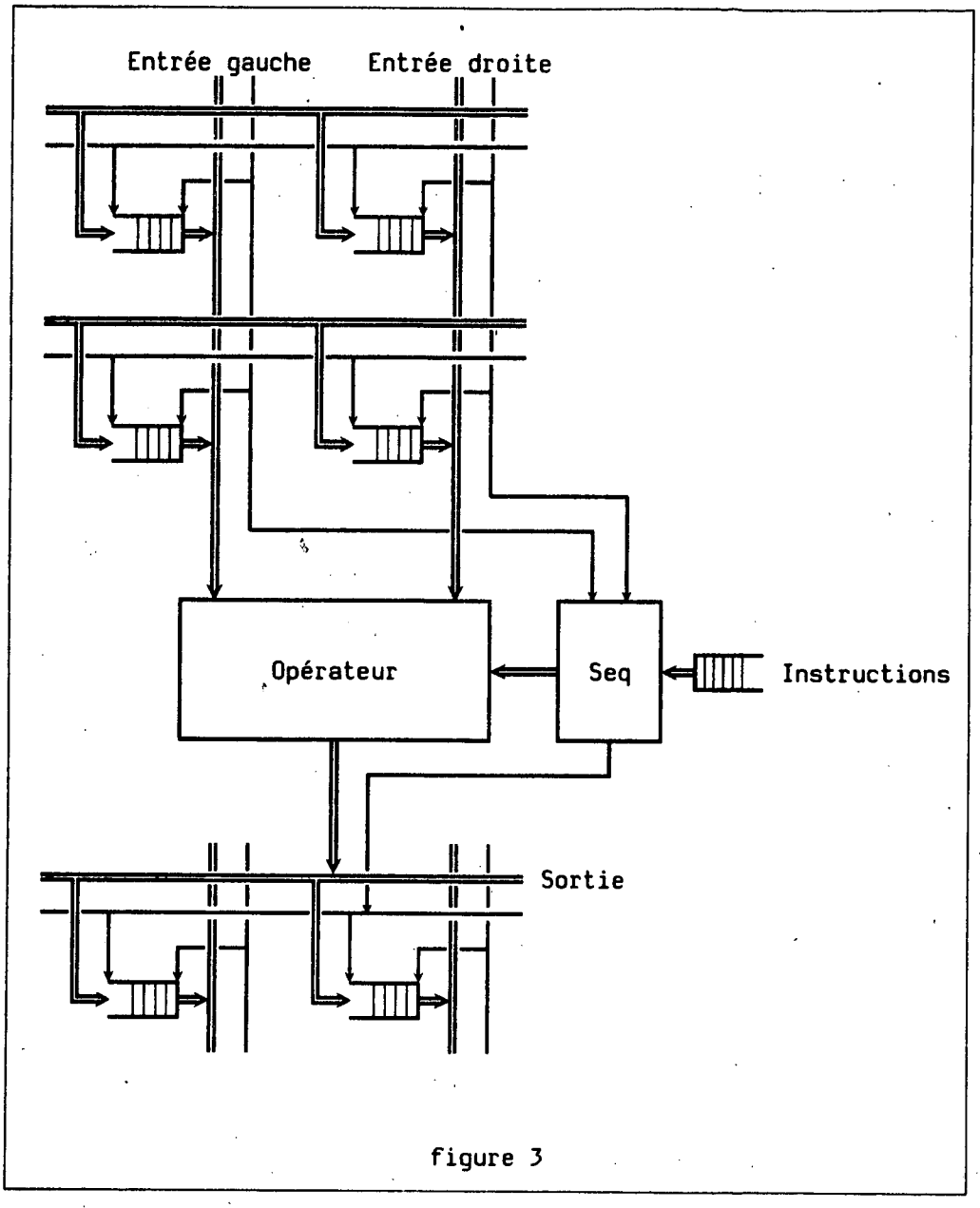

figure 3

- phase 1 : Extraction de la prochaine instruction de la FIFO d'instructions. Cette phase est retardée tant que la FIFO d'instructions est vide.

- phase 2 : Décodage de l'instruction.

- phase 3 : Extraction des opérandes. Cette phase provoque la sélection des FIFO sources et l'extraction des données. Cette phase est terminée lorsque tous les opérandes ont été reçus.

- phase 4 : Exécution de l'instruction par l'opérateur. Cette phase peut durer plusieurs cycles.

- phase 5 : Sélection de la FIFO destinatrice et émission du résultat.

Ces phases sont relativement indépendantes, et des phases d'instructions successives peuvent être actives simultanément. Cette possibilité est nécessaire pour permettre une utilisation optimale d'opérateurs pipeline. La synchronisation de cet opérateur avec son environnement se fait par les données (d'où le nom DSPA du processeur). Dans une mise en oeuvre réelle, les FIFO sont de longueur limitée, ce qui imposeune contrainte de disponibilité sur la phase 5 précédente. Plusieurs solutions existent pour assurer que les données émises peuvent être mémorisées dans les FIFO destinatrices. La première solution consiste à vérifier que la FIFO n'est pas pleine pendant la phase 5 d'émission. Mais, lorsque le résultat ne peut être émis, il faut arrêter non seulement le séquencement de cette instruction mais également le séquencement des instructions suivantes. Ceci n'est pas toujours possible sur les opérateurs pipelines. Une deuxième solution consiste à vérifier la disponibilité de la FIFO destinatrice pendant la phase 2 ou la phase 3. Cette vérification peut se faire de deux manières. Soit que le signal FIFO pleine est déclenché lorsque la FIFO de longeur M+N contient au moins un certain nombre N de valeurs. Il suffit alors de bloquer la phase 3 si le signal FIFO pleine de la FIFO destinatrice est levé, les M emplacements supplémentaires garantissant que les résultats des M opérations en cours sur l'opérateur pourront être mémorisés. Ce blocage entraîne également le blocage des instructions suivantes, mais l'exécution des instructions précédentes peut continuer sur l'unité fonctionnelle. Une autre solution consiste à émettre un signal de réservation vers la FIFO destinatrice pendant la phase 2. Ce signal provoque la mise à jour du signal FIFO pleine et permet donc de garantir la présence d'un emplacement au moment de l'émission du résultat. Nous verrons par la suite que l'émission d'un tel signal permet des mises en oeuvres dans lesquelles certaines ressources doivent être partagées par les unités fonctionnelles.

le DSPA, un pipeline synchronisé par les données

3 la mémoire locale

La présence de registres dans les architectures classiques permet l'utilisation multiple des données internes d'un processeur, dans les multiplications de complexes par exemple, et de limiter les accès à la mémoire globale sur des scalaires fréquemment utilisés comme les compteurs de boucles, certaines bases de calcul d'adresses etc... La présence de FIFOs d'échange entre les unités fonctionnelles interdit de tels accès multiples bien que certains cas fréquents comme les multiplications de complexes puissent être traités directement au niveau des opérateurs. Chaque unité fonctionnelle d'un DSPA exécutant ses instructions dans l'ordre où elles ont été extraites du programmes par le séquenceur global, rien n'interdit la mémorisation de données localement à une unité fonctionnelle, et donc son utilisation multiple. La présence d'une unité mémoire locale nous semble primordiale, surtout dans un contexte multiprocesseur. Nous traitons ici la cas d'une mémoire locale de stockage de scalaires sur laquelle seules des instructions de lecture et d'écriture à des adresses absolues (connues à la compilation) sont prévues. Cette unité fonctionnelle joue le rôle des registres d'une machine classique.

Cette unité fonctionnelle M1 a une seule entrée logique des données à écrire M1 (l'adresse étant codée dans l'instruction) et une sortie logique des données lues. Une instruction de lecture sur la mémoire locale a le format

M1 | lire | adresse | destination

et une instruction d'écriture le format

M1 | écrire | adresse | source

Dans une première approche, nous considérons que les accès à cette mémoire locale sont effectuées dans l'ordre où les instructions sont extraites du programme, ce qui permet de garantir la cohérence des données internes. Un mécanisme de doublage des écritures par les lectures semblable à celui décrit pour l'unité mémoire (voir §4-b) peut être introduit. Dépendant des besoins internes au processeur, des instructions plus spécifiques peuvent être définies, comme l'instruction de sauvegarde d'une valeur

M1 | sauver | adresse | source | destination

Cette instruction provoque l'écriture de la donnée source dans la mémoire locale et son émission vers une autre unité fonctionnelle, et permet de coder plus efficacement les utilisations multiples des données.

4 la mémoire

L'unité fonctionnelle mémoire doit être soigneusement définie dans une architecture multiprocesseur à couplage serré. Elle doit en effet

- permettre la gestion d'une mémoire entrelacée et partagée,
- permettre le doublage des écritures par les lectures tout en contrôlant les dépendances,
- donner une signification aux accès parallèles dans une architecture multiprocesseur.

Nous traitons ici le cas d'un monoprocesseur. Les contraintes liées aux multiprocesseurs sont traitées en [Jégou 86a].

Comme sur la mémoire locale, les deux instructions de base de la mémoire principale sont *lire* et *écrire*. Mais, ici, l'adresse n'est plus codée dans l'instruction, mais reçue d'une unité fonctionnelle. Le cas de l'adressage absolu est, en général, traité en considérant que l'unité fonctionnelle séquenceur émet les valeurs en extension des instructions vers les entrées logiques des unités fonctionnelles. L'unité fonctionnelle mémoire M a donc deux entrées logiques : l'entrée dM des données à écrire et l'entrée aM des adresses. Elle possède une sortie logique M des données lues.

L'instruction *lire* a le format de base

M | lire | source adresse | destinataire

L'instruction *écrire* a le format de base

M | écrire | source adresse | source donnée

Pour obtenir un séquencement rapide des programmes, le codage des instructions fréquemment utilisées doit être le plus compact possible. Par exemple, l'accès aux éléments consécutifs d'un vecteur rangé linéairement peut se faire par incrémentation d'un registre d'adresse par un registre d'incrément à chaque accès. Ces calculs peuvent être réalisé par une unité fonctionnelle de calcul d'adresses en mémorisant l'adresse de base et l'incrément dans la mémoire locale. Mais, dans ce cas, il faut coder les instructions d'accès aux registres, les instructions de l'unité de calcul des adresses et les instructions d'accès à la mémoire. Ce type d'accès est fréquent dans les algorithmes qui nous intéressent, l'adressage linéaire étant recherché dans les langages de type Hellena ([Jégou 86b], nous faisons par la suite référence à ce langage qui a servi de base à l'étude des besoins sur les architectures). Un codage très compact est obtenu en gérant ces descripteurs de vecteurs directement dans l'unité fonctionnelle mémoire. Une gestion très simple des descripteurs internes aux unités fonctionnelles est rendue possible par le séquencement strict des instructions. L'unité fonctionnelle mémoire d'un DSPA possède un certain nombre de paires de registres internes et un additionneur. Un accès postincrémenté fait référence à un numéro de descripteur interne, provoque un accès à la mémoire à l'adresse contenue dans le registre d'adresse et remet à jour ce registre en ajoutant la valeur de l'incrément pour le prochain accès. Les instructions d'accès postincrémenté ont le format suivant :

M | lirepic | descripteur | destinataire
M | écrirepic | descripteur | source donnée

L'entrée logique d'adresses n'est pas utilisée dans ces instructions. Ces descripteurs sont locaux à l'unité mémoire et tout accès direct par les autres unités fonctionnelles doit être interdit pour respecter la cohérence des adresses décrites. Il faut donc prévoir des instructions d'initialisation de ces descripteurs sur l'unité fonctionnelle mémoire.

M | initialiser base | descripteur | source adresse
M | initialiser raison | descripteur | source raison

Ces deux instructions prennent la valeur initiale sur l'entrée logique d'adresses. Les instructions étant exécutées dans l'ordre de séquencement du programme, l'exécution d'une instruction d'initialisation ne peut donc pas provoquer la perte de descripteurs utiles.

D'autres instructions (figure 4) d'accès fréquents peuvent être également définies. Les instructions empiler et dépiler font référence implicitement

M	empiler	source donnée		
M	dépiler	destinataire		
M	lire indirect	descripteur	destinataire	
M	écrire indirect	descripteur	source donnée	
M	lire basé	descripteur	source adresse	destinataire
M	écrire basé	descripteur	source adresse	source donnée

figure 4

à un registre d'adresse de sommet de pile local à l'unité mémoire. Les instructions de lecture et d'écriture indirectes peuvent n'utiliser que la partie adresse des descripteurs, ce qui permet de traiter le cas où un même vecteur est accédé plusieurs fois dans une même instruction vectorielle. Les instructions d'adressage basé ajoutent l'adresse source à un registre d'adresse interne pour calculer les adresses effectives.

En fait, toutes les instructions d'accès classiques à la mémoire peuvent être prévues. Il suffit de prévoir les registres internes et les opérateurs localement à l'unité fonctionnelle. Notons par ailleurs que, le fait que le décodage des instructions soit local à l'unité fonctionnelle permet un formatage très libre de ces instructions.

Comme toutes les unités fonctionnelles d'un DSPA, l'unité fonctionnelle mémoire exécute ses instructions dans l'ordre de séquencement global, et émet les données lues dans l'ordre correspondant vers les entrées logiques des unités fonctionnelles du processeur. Mais, le séquencement interne des requêtes doit être soigneusement étudié pour permettre une utilisation optimale des mémoires parallèles entrelacées et

le dépassement contrôlé des écritures physiques par les lectures, contraintes que nous avons exposé en introduction.

a) gestion d'une mémoire parallèle

Dans une première approche, nous considérons que tous les bancs parallèles commutent avec un même délai *d*. Une première solution consiste alors à exécuter une instruction d'accès lorsque le banc requis est disponible. Un séquencement possible de l'unité fonctionnelle mémoire est décrit dans la figure 5. Ces phases étant relativement indépendantes, plusieurs phases

phase 1 ↦ extraction de l'instruction ;
phase 2 ↦ extraction et calcul de l'adresse
extraction de la donnée à écrire si besoin ;
phase 3 ↦ attente disponibilité du banc ;
phase 4 ↦ initialisation de la requête ;
phase 5 ↦ émission du résultat si besoin.

figure 5

d'instructions successives peuvent être actives simultanément. Les phases 1, 2 et 3 peuvent être de durée variable. Par contre, la phase 5 a lieu *d* cycles après la phase 4. Cette phase 5 n'a pas lieu pour les requêtes d'écriture. La cohérence de la mémoire est respectée lorsque les phases 4 sont initialisées dans l'ordre d'extraction des instructions. Le shémas fonctionnel d'une telle mémoire est décrit par la figure 6. Le séquencement de cette mémoire est proche du séquencement adopté pour les instructions d'accès à la mémoire du Cray 1. Comme l'ordre de commutation des bancs correspond à l'ordre des instructions et que tous les bancs commutent avec un même délai, les données lues sont émises directement sur la sortie de l'UF mémoire vers les entrés des unités fonctionnelles du DSPA. Mais, le comportement d'une telle mémoire n'est pas optimal. Une attente dans l'initialisation de la phase 4 d'une requête entraîne un retard dans l'initialisation de cette même phase des autres requêtes présentes qui, elles, pourraient s'adresser à des bancs disponibles. De plus, dans une configuration multiprocesseur, ces bancs ne peuvent se partager un même bus de sortie pour permettre des accès parallèles. Comme

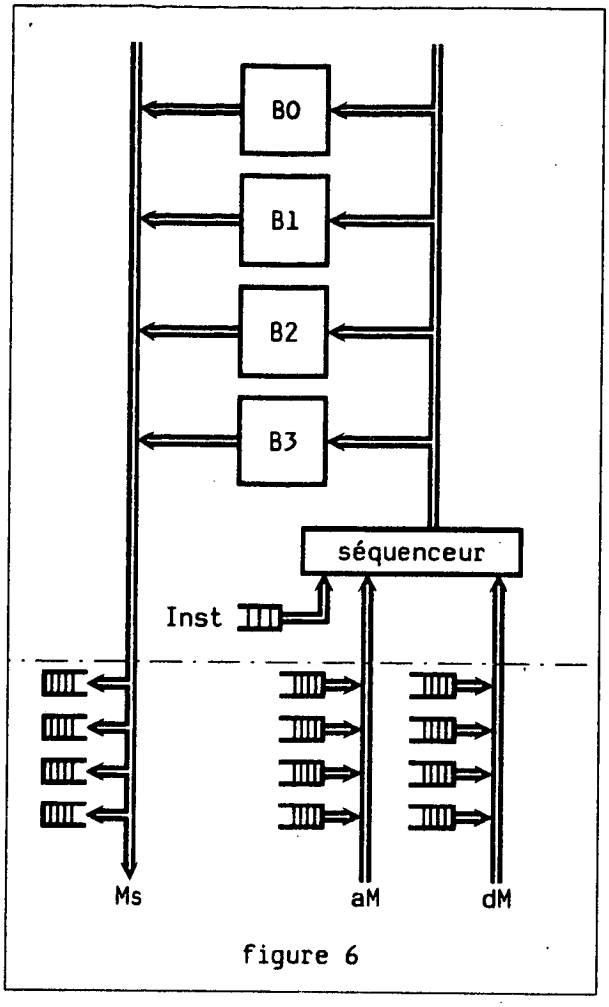

figure 6

nous l'avons fait pour les séquencement des unités fonctionnelles d'un DSPA, il est possible de dissocier les séquencements de chaque banc mémoire. Pour ce faire, nous associons une FIFO de requêtes à chaque banc. La phase 3 du séquencement de l'unité fonctionnelle mémoire consiste alors à émettre la requête vers la FIFO de requêtes du banc destinataire et ne provoque donc aucune attente (à moins que cette FIFO ne soit pleine). Mais, les bancs étant indépendants, l'ordre de commutation des bancs ne correspond plus à l'ordre de séquencement des instructions globales : les résultats doivent être réordonnés avant d'être émis vers les destinataires. Pour celà, une FIFO de résultat est associée à chaque banc, et une unité de réordonnement est introduite. Cette unité de réordonnement reçoit dans une

FIFO de commandes les informations de réordonnement (la suite des numéros de bancs auxquels les requêtes ont été émises et les entrées logiques des unités fonctionnelles destinatrices correspondantes). Ces informations sont rangées dans cette FIFO pendant la phase 3 du séquencement de l'unité fonctionnelle mémoire.

La figure 7 décrit une telle unité fonctionnelle. Le séquenceur extrait

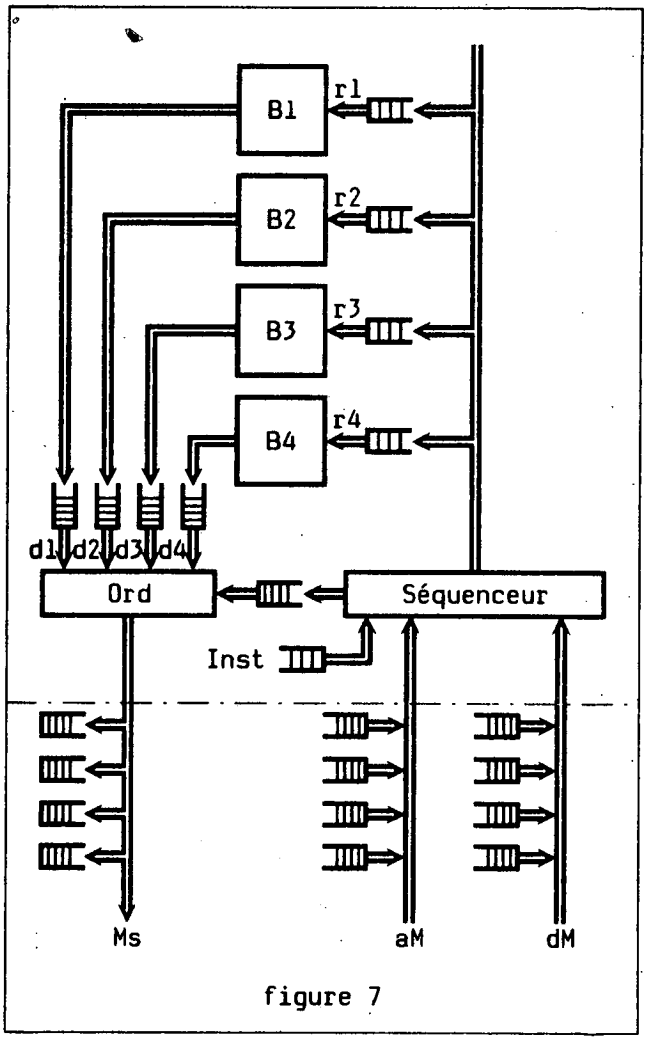

figure 7

ses instructions de la FIFO Inst, les adresses sur l'entrée logique aM et les données sur l'entrée logique dM. Rappelons que plusieurs FIFOs sont associées à chacune de ces entrées logiques. Le séquenceur émet vers les

FIFOs r1, ..., rn les requêtes des bancs B1, ..., Bn. Ces bancs rangent les résultats des lectures dans les FIFOs d1, ..., dn. L'unité de réordonnement reçoit du séquenceur des informations constituées de couples banc-destinataire dans la FIFO C et des FIFOs d1, ..., dn les données à émettre vers les entrées des unités fonctionnelles du processeur. Les données lues sont donc émises dans l'ordre de séquencement des instructions de la mémoire.

Le comportement d'une telle mémoire est proche de l'optimal car l'occupation d'un banc n'arrête pas le traitement des requêtes suivantes. Dans une architecture multiprocesseur, ces bancs mémoires sont partagés par l'ensemble des processeurs [Jégou 86a]. Chaque processeur possède son propre séquenceur d'unité fonctionnelle mémoire, sa propre unité de réordonnement et son propre ensemble de FIFOs de réception des données lues d1, ..., dn. Pour *m* bancs et *p* processeurs, il y a donc *mp* FIFOs de résultats. Une donnée émise par le banc *i* vers le processeur *j* sur l'entrée *i* du réseau d'interconnexion est rangée dans la FIFO *di* du groupe de FIFOs associé au processeur *j* et donc à la sortie *j* du réseau. Par contre, dans une première approche, *m* FIFOs R de requêtes (une par banc) suffisent à condition que le numéro du processeur qui a émis la requête lui soit associé pour permettre le routage du résultat. En fait, cette structuration dépend du type de réseau d'interconnexion utilisé, du type de requêtes prévues sur la mémoire, et du fonctionnement du mécanisme de traitement du doublage des écritures par les lectures. Ces divers paramètres sont traités en [Jégou 86a]. Notons enfin que, d'une part, dans un schémas multiprocesseur, chaque mémoire peut être elle même décomposée en bancs auquel cas le schémas de contrôle proposé peut être répété à chaque niveau, d'autre part l'utilisation d'un mécanisme de réordonnement n'impose plus que les durées de commutation des bancs soient les mêmes. Il est donc possible de construire des mémoires sur lesquelles l'accès à certaines adresses est plus rapide.

b) traitement du doublement des écritures par les lectures

Nous avons indiqué en introduction que les requêtes de lectures devaient pouvoir doubler les requêtes d'écriture sur la mémoire sous peine de vider tous les pipelines à chaque itération des boucles. Ces doublements doivent être contrôlés pour respecter les dépendances du type

lecture après écriture. Il suffit que les écritures soient faites dans l'ordre et que les écritures ne puissent pas doubler les lectures pour que la détection des dépendances du type écriture après écriture ou écriture après lecture soient respectées.

Le principe du traitement des requêtes à la mémoire avec doublement des écritures par les lectures et détection des aléas est le suivant :

↦ Les adresses sont calculées dans l'ordre.

↦ Le traitement des requêtes d'écriture se fait en deux phases. La première consiste à calculer l'adresse d'écriture et à la mémoriser. La deuxième phase d'écriture proprement dite est exécutée lorsque la donnée à écrire est présente. Le séquencement des instructions d'écriture en mémoire n'est donc pas lié à la présence des données à écrire. Le débit de séquencement des instructions de l'unité fonctionnelle mémoire n'est limité que par la présence des instructions et des adresses correspondantes.

↦ A chaque fois que l'adresse correspondant à une requête est calculée, elle est comparée aux adresses des écritures en attente. En cas d'égalité, le traitement de l'instruction correspondante est suspendu (respect de l'ordre des accès à une même adresse) jusqu'à ce que la requête en attente d'écriture ait été traitée.

Ce mécanisme de traitement des doublements contrôlés peut être installé localement à un processeur dans l'unité fonctionnelle mémoire. Mais dans ce cas, il ne permet pas de traiter les aléas interprocesseurs, limitation qui n'est pas forcément gênante sur un multiprocesseur MIMD, mais qui entraîne l'introduction d'instructions de synchronisation explicite coûteuses sur une machine fortement couplée. Ce mécanisme doit donc être partagé et placé au niveau de la mémoire partagée. Il suffit de constater que des aléas de lecture après écriture ne peuvent apparaître qu'entre des requêtes destinées à un même banc pour décentraliser de tels mécanismes au niveau des bancs physiques (contraintes technologiques minimales bien que cette solution entraîne une duplication du mécanisme). Cette décentralisation permet d'ailleurs de ne bloquer que les requêtes destinées à un même banc lorsqu'une dépendance est détectée. Par exemple, soit une architecture multiprocesseur composée de 16 processeurs, 16

mémoires, chacune de ces mémoires étant elle-même composée de huit bancs entrelacés :

↦ Un seul mécanisme implanté au niveau de la mémoire doit pouvoir traiter 16 requêtes (une par processeur) par cycle.

↦ Un mécanisme implanté sur chaque mémoire (16 mécanismes en tout) doit traiter une requête par cycle.

↦ Un mécanisme par banc physique (128 mécanismes en tout) doit traiter une requête tous les huit cycles (au moins).

De plus, le nombre d'adresses d'écriture à mémoriser (et donc à comparer à chaque cycle) est directement proportionnel au débit demandé au mécanisme de détection des aléas. Par exemple, en partant d'une base de 16 adresses mémorisées par banc physique, il faut 128 adresses par mémoire et 2048 pour un mécanisme global à la mémoire. Le choix d'implantation d'un tel mécanisme dépend des possibilités technologiques de réalisation. Dans la suite, nous considérons le cas du traitement des doublements et des aléas au niveau des bancs physiques.

Un tel mécanisme peut être réalisé à partir d'une mémoire associative, ce qui est réaliste lorsque le nombre d'adresses à mémoriser est faible. Des implantations simplifiées peuvent être réalisées sans véritablement dégrader les performances du calculateur. Par exemple, il suffit de réaliser un ensemble de n registres et d'associer chaque adresse mémoire (du banc) à l'un de ces registres par une certaine fonction f. Un marqueur indique si chacun de ces registres contient une adresse ou non (marqueur d'occupation). L'adresse A de chaque requête (lecture ou écriture) est comparée au contenu du registre $f(A)$. En cas d'égalité, un aléas est détecté et le séquencement est interrompu. Le traitement d'une requête d'écriture est également interrompu si le registre correspondant est déjà occupé. Le défaut de ce mécanisme est que, lorsque le traitement d'une requête d'écriture doit être suspendu pour cause d'occupation du registre correspondant, le traitement de toutes les requêtes suivantes doit également être suspendu (l'adresse de la requêtes suspendue n'ayant pas été mémorisée, il n'est pas possible de détecter des aléas sur cette adresse). Le choix de la fonction f et un nombre suffisemment grand de registres

limitent ces désagréments. Il est également possible de doubler ce mécanisme pour diminuer la probabilité d'un blocage abusif des requêtes d'écritures. Par contre, ce mécanisme simplifié n'entraîne pas de suspension "abusive" des requêtes de lecture. Mais il impose des dépendances artificielles entre des requêtes indépendantes (les requêtes d'écriture qui se partagent un même registre). Pour que ces dépendances artificielles ne bloquent définitivement le séquencement des requêtes, il faut éviter de coder une instruction d'écriture mémoire avant les instructions de lecture des opérandes de calcul de la donnée à écrire. Cette contrainte nous semble naturelle dans le cas général. Nous rappelons que, lorsque ce mécanisme est localisé au niveau des bancs physiques, la présence d'une dépendance ne suspend que le séquencement de ce banc physique.

Un deuxième facteur à prendre en compte dans le choix d'un tel mécanisme est l'association d'une donnée à écrire à son adresse. La solution la plus simple consiste à émettre les données à écrire dans l'ordre où leurs adresses ont été émises. Dans une architecture monoprocesseur, il est alors possible de gérer la mémoire associative cycliquement et donc de simplifier à la fois la gestion de cette mémoire et l'association d'une donnée à l'adresse correspondante. Une autre solution consiste à réaliser cette association à travers une FIFO des entrées utilisées de la mémoire associative ou du H-code partiel, ce qui permet de réaliser une FIFO des entrées libres dans le cas des mémoires associatives. Ces solutions permettent de n'utiliser que des circuits existants. Les deux fonctionnalités à mettre en oeuvre, comparaison aux adresses d'écriture en attente et association des données à écrire à leurs adresses pourraient être intégrées dans un seul circuit qui combinerait une gestion en FIFO des adresses en attente et une comparaison des adresses de lecture aux adresses en attente de la manière suivante. Le traitement d'une requête d'écriture se limite au rangement de l'adresse en queue de la file d'attente. Il n'est pas nécessaire de vérifier si cette adresse existe déjà. Les données à écrire arrivant dans l'ordre des requêtes, il suffit d'écrire la donnée en tête de la FIFO des données à l'adresse en tête de la FIFO des adresses. Par contre, le traitement d'une requête de lecture est accompagné de la comparaison de son adresse aux adresses mémorisées dans la FIFO des adresses (fonctionnalité spécifique à cette FIFO). Dans une architecture multiprocesseurs, il ne peut y avoir de relation entre les

données émises par des processeurs différents. Dans ce cas, un système d'association plus complexe que nous traitons en [Jégou 86a] doit être mis en oeuvre.

Lorsqu'une requête de lecture est suspendue pour cause de dépendance, elle peut, en général, être traitée dès que la donnée à écrire est disponible : il n'est pas nécessaire de faire suivre le cycle d'écriture dans le banc par un cycle de lecture à la même adresse. Cette possibilité peut être étendue en conservant les données reçues en écriture dans la mémoire associative tant que l'entrée correspondante n'est pas réallouée. Cette possibilité permet d'accélérer l'accès aux adresses récemment écrites. Mais, pour des raisons technologiques, ces possibilités ne peuvent dans la pratiques être implantées que lorsque le mécanisme de doublement est localisé au niveau des bancs physiques.

En conclusion, l'unité fonctionnelle mémoire d'un DSPA est décomposée en deux parties. L'unité fonctionnelle proprement dite est localisée dans le processeur et possède un séquenceur/décodeur d'instructions Seq, une unité de réordonnement des données lues Ord, et une unité d'émission des données à écrire Ecr (figure 8). Le séquenceur/décodeur reçoit les instructions à exécuter et est connecté aux FIFOs de l'entrée logique d'adresses aM. Ce séquenceur gère également des registres internes d'adressage postincrémenté, basé, indirect ou sur pile. Ce séquenceur émet les requêtes vers la mémoire proprement dite, les informations de réordonnement vers l'unité de réordonnement, et les informations nécessaires vers l'unité de gestion des écritures. La sortie logique Ms de l'unité fonctionnelle mémoire est associée à l'unité de réordonnement. L'entrée logique de données dM de la mémoire est associée à l'unité d'écriture. Cette dernière unité reçoit du séquenceur des couples source-banc nécessaires à la récupération des données produites par les unités fonctionnelles productrices des données à écrire et à l'émission de ces données vers la mémoire. Dans l'architecture multiprocesseur que nous décrivons par la suite, le séquenceur émet ses requêtes à travers un réseau de requêtes, l'unité de réordonnement reçoit ses données du réseau de lecture et l'unité d'écriture émet ses données sur le réseau d'écriture.

5 l'unité de calculs sur les entiers

Cette unité fonctionnelle est utilisée principalement pour le calcul des adresses. Elle peut être structurée comme l'opérateur flottant décrit en §2. Mais, comme nous l'avons fait pour les différents types d'adressage fréquents sur la mémoire, il est intéressant d'analyser les opérations fréquentes de préparation d'adresses. Lorsque seules des instructions scalaires sont disponibles, les descripteurs prévus dans l'UF mémoire peuvent suffire pour les adressages linéaires. Mais, nous verrons en §9 que rien n'interdit l'introduction d'instructions vectorielles sur une

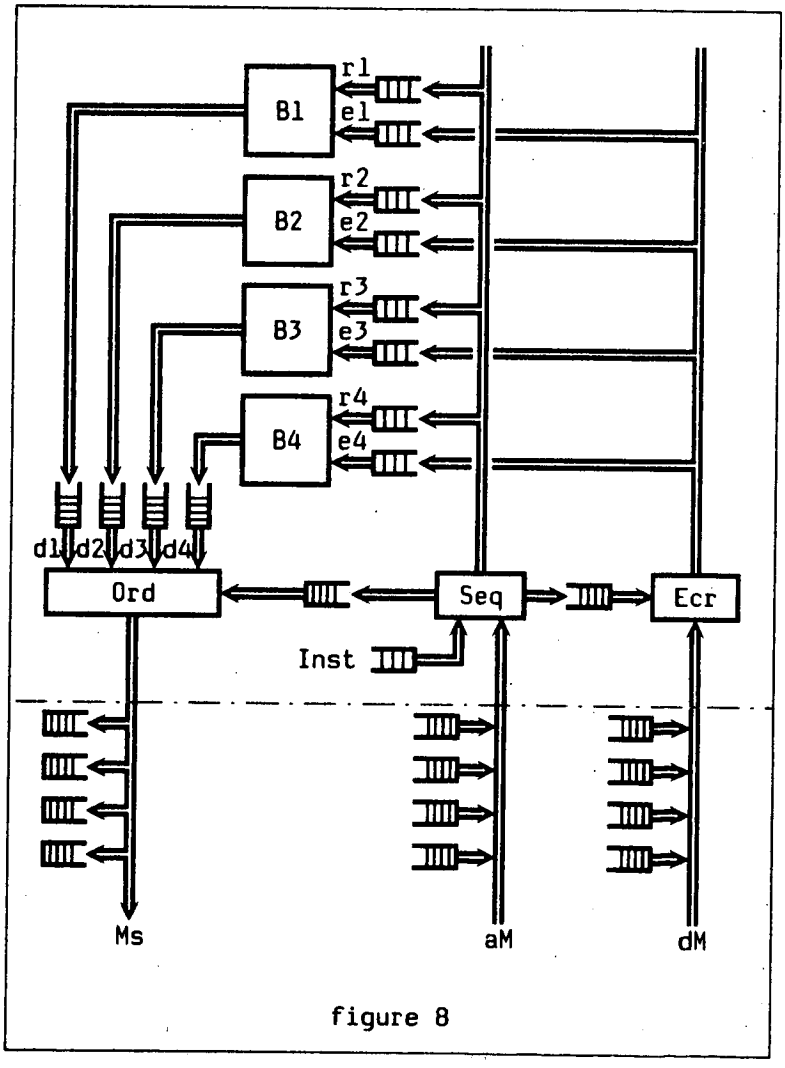

figure 8

architecture DSPA. Dans ce cas, les calculs des adresses des éléments des vecteurs peuvent être pris en charge par l'unité fonctionnelle mémoire. Mais ces descripteurs de vecteurs doivent pouvoir être calculés rapidement. L'un des rôles de cette unité fonctionnelle entière est de préparer ces descripteurs. Pour celà, l'unité fonctionnelle entière peut être dotée de descripteurs et d'instructions spécifiques de génération de séquences linéaires de valeurs. Les résultats ne sont pas nécessairement émis vers l'unité fonctionnelle mémoire, ce qui permet par exemple la gestion d'indices de boucles, ou la production de toute séquence linéaire qui peut apparaître dans un programme. Cette unité fonctionnelle doit être dotée d'un additionneur entier, d'un multiplieur pour accélérer les calculs d'adresses des éléments des tableaux (il faut en général une multiplication et une addition par indice lorsque l'accès n'est pas linéaire). D'autres instructions comme l'encodage de priorité, le comptage du nombre de bits à un d'un mot, le calcul des adresses du perfect shuffle d'un vecteur etc... sont également utiles dans certains accès aux données. En conclusion, cette unité fonctionnelle se présente comme une unité de calcul numérique du DSPA (deux entrées logiques gauche et droite et une sortie logique) et peut être enrichie par des descripteurs internes favorisant la production de certaines séquences de valeurs. Dans une version simplifiée d'un processeur DSPA, il est possible de supprimer les descripteurs de l'unité fonctionnelle mémoire et de réaliser les calculs d'adresses linéaires, basées, etc sur cette unité entière.

6 l'unité de séquencement

Nous avons vu que le séquencement d'un DSPA se limite au décodage des numéros d'unité fonctionnelle des instructions, à l'émission des ces instructions vers les FIFOs d'instructions des unités fonctionnelles désignées, et à la détection des instructions à destination du séquenceur. Les instructions du séquenceur de programme devant être traitées immédiatement, la présence d'une FIFO d'instructions n'est pas utile. Ces instructions peuvent être classées en trois catégories :

O les instructions de rupture de séquence.

O les instructions d'état.

O le traitement des valeurs en extension.

Les instructions de rupture de séquence sont les sauts, les sauts conditionnels, les instructions d'appel et de retour de sous programmes. Ces instructions font référence à l'unité fonctionnelle source de l'adresse de saut et éventuellement à l'unité fonctionnelle qui produit la condition dans le cas des sauts conditionnels. Les instructions d'appel de procédures émettent l'adresse de retour sur la sortie logique de l'unité fonctionnelle séquenceur. Ces adresses peuvent ainsi être émises, soit vers la mémoire locale, la mémoire globale, ou directement sur l'entrée de données du séquenceur dans le cas de routines courtes. La présence d'une entrée logique des adresses de saut permet d'exécuter des sauts à des adresses calculées à l'exécution. Cette possibilité est intéressante pour la traduction de l'instruction case de nombreux langages, et facilite la programmation d'automates. Le traitement d'une instruction de rupture de séquence provoque le calcul d'une nouvelle valeur du compteur ordinal où le séquencement peut reprendre. Comme nous l'avons fait pour l'unité fonctionnelle mémoire, cette unité peut être dotée de descripteurs permettant d'accélérer certaines séquences de traitements. Par exemple, on peut prévoir des registres de comptage de boucles et les instructions d'itération correspondantes.

Les instructions d'état peuvent agir sur des informations globales à la machine. C'est le cas des masques d'instructions vectorielles que nous décrivons en §10.

Dans les architectures classiques, les constantes d'un programmes comme les adresses absolues de la mémoire sont généralement codées en extension des instructions, dans le mot suivant, par exemple. Dans le cas du DSPA, il n'est pas possible de coder de telles constantes dans une partie paramètre des instructions des unités fonctionnelles. Un tel codage entraînerait soit une largeur d'instruction prohibitive, soit une complexité supplémentaire du séquencement. Ce problème est résolu en définissant une instruction d'extension sur le séquenceur. Cette instruction a le format suivant

Séquenceur | extension | destinataire

Lorsqu'une telle instruction est rencontrée, le séquencement implicite est interrompu (c'est une instruction à destination du séquenceur), le mot suivant (ou deux mots pour les doubles mots) est extrait par le séquenceur et émis vers la FIFO destinataire, et le séquencement reprend à la suite. Bien qu'elles puissent être traitées directement dans les cas de branchements à des adresses absolues, une telle stratégie peut également être adoptée pour les constantes du séquenceur (adresses de saut).

Nous avons décrit les principes de base du fonctionnement de l'unité fonctionnelle de séquencement. Nous reviendrons en §9 sur les problèmes liés au fonctionnement en mode vectoriel, en §12 sur les traitements des erreurs et des interruptions et en [Jégou 86a] sur le fonctionnement multiprocesseur fortement couplé.

Le fonctionnement décrit permet un séquencement rapide d'une suite d'instructions. Pour le séquencement rapide des instructions de rupture de séquence, toutes les solutions classiques peuvent être adoptées. Les instructions peuvent être rangés dans la mémoire principale (ce qui impose un cadrage des instructions sur des unités adressables de cette mémoire), extraites d'un ou plusieurs caches d'instructions, ou être rangés dans une mémoire spécifique pour limiter le débit demandé à la mémoire principale. On peut également prévoir des instructions de sauts différés, des instructions de préparation de sauts (qui provoquent le chargement du cache d'instructions) ou des systèmes de prédiction de saut. Une solution plus spécifique du DSPA consiste à spécialiser l'une des entrées logiques de cette unité fonctionnelle dans la réception des adresses de branchements. Il suffit alors d'émettre ces adresses le plus tôt possible, l'arrivée d'une adresse dans l'une des FIFOs associées à cette entrée logique provoquant soit un préchargement d'un cache d'instructions, soit une lecture non prioritaire d'une ou plusieurs instructions à partir de cette adresse.

7 comportement général du DSPA

Notre objectif était de définir un processeur capable d'exécuter rapidement du code séquentiel. Pour celà, un séquencement rapide des instructions des programmes est nécessaire. Cet objectif a été atteint en

limitant au strict minimum les opérations de décodage nécessaires pendant cette phase de séquencement. En reportant le décodage des instructions au niveau de chaque unité fonctionnelle et en évitant l'utilisation de registres pour réaliser les échanges de données, nous avons rendu possible le séquencement et le décodage parallèle des instructions (une par unité fonctionnelle). Ces décodeurs sont spécialisés, et donc plus simples à réaliser. Dans de nombreux cas, ces décodeurs peuvent également être pipeline, ce qui augmente encore le parallélisme global du décodage. La synchronisation par les données simplifie grandement l'utilisation d'unités fonctionnelles de débits différents, ainsi que la gestion d'unités fonctionnelles dont le temps de réponse à une instruction ne peut être prédit ni au séquencement, ni au décodage. On pourrait même considérer que chaque unité fonctionnelle possède sa propre horloge interne, l'asynchronisme étant totalement géré au niveau des FIFOs. Cette souplesse a permis, par exemple, de déporter le système de gestion du doublement des écritures par les écritures au niveau de la mémoire.

L'algorithme séquentiel de multiplication de matrice creuse par un vecteur plein suivant peut être exécuté avec un bon remplissage des pipelines.

```
pour_tout i sequentiellement dans [1,n] :
   a(l(i)) := a(l(i)) + v(i)*b(c(i)) ;
fin_pour_tout ;
```

Dans le cas le plus défavorable, nous supposons que les raisons d'accès aux vecteurs de ce programme sont différents de 1, ce qui entraîne une multiplication pour calculer l'adresse de chaque élément. L'accès aux vecteurs l, c et v peut être fait par des instructions de postincrémentation sur les descripteurs Dl, Dc et Dv de l'unité fonctionnelle mémoire. Par contre, les adresses de base de a et de b ainsi que leurs raisons d'accès sont rangées en mémoire locale aux adresses Aa, Ra, Ab et Rb. En supposant de plus que le compilateur n'a pas détecté la présence de a(l(i)) deux fois dans l'instruction (descripteur Dl1 et Dl2 identiques, adresses Aa1 et Aa2 identiques et raisons Ra1 et Ra2 identiques), la figure 9 donne traduction directe de la boucle principale de cet algorithme sur un DSPA. Cette séquence peut être produite très

Ml	lire, n	dS
Sq	initBS, Ri, Ml	
Ml	lire, Rb	Alg
M	lpic, Dc	Ald
Ml	lire, Ra2	Alg
M	lpic, D12	Ald
Ml	lire, Ab	Alg
Al	*, Ml, M	Ald
Ml	lire, Aa2	Alg
Al	*, Ml, M	Ald
Al	+, Ml, Al	aM
Ml	lire, Ra1	Alg
M	lpic, D11	Ald
Al	+, Ml, Al	aM
M	lpic, Dv	*fg
M	lire, Al	*fd
Ml	lire, Aa1	Alg
Al	*, Ml, M	Ald
M	lire, Al	+fg
*f	*, M, M	+fd
Al	+, Ml, Al	aM
+f	+, M, *f	dM
M	écrire, Al, +f	
Sq	itère,Ri	

figure 9

simplement par un procédé que nous détaillons en §8. Les unités fonctionnelles utilisées et leurs instructions sont :

→ M : mémoire principale qui possède les entrées logiques d'adresses aM et de données dM. L'instruction lire fait référence à l'une des FIFOs de l'entrée logique aM. L'instruction écrire prend l'adresse sur l'entrée logique aM et la donnée sur l'entrée logique dM. L'instruction de lecture avec postincrémentation d'adresse lpic fait référence à un descripteur

interne. Ce descripteur doit avoir été initialisé avant l'exécution de la boucle.

↦ Ml : mémoire locale qui possède une seule entrée logique non utilisée dans cet exemple. Les adresses sont codées dans l'instruction (c'est une petite mémoire).

↦ Al : unité de calcul sur des entiers. Cette unité est chargée dans cet exemple d'effectuer les calculs d'adresses non linéaires (multiplications et additions). Cette unité possède deux entrées gauche Alg et droite Ald.

↦ *f : unité multiplieur flottant possède les entrées logiques gauche *fg et droite *fd.

↦ +f : unité additionneur flottant possède les entrées logiques gauche +fg et droite +fd.

↦ Sq : unité de séquencement. Nous utilisons ici l'instruction itère qui provoque la décrémentation d'un compteur interne et un saut éventuel à l'instruction de début de boucle dont l'adresse est également mémorisée dans un registre interne Ri. La boucle scalaire est initialisée par l'instruction initBS qui prend le nombre d'itérations sur l'entrée logique dS, range cette valeur ainsi que l'adresse de l'instruction suivante (la première de la boucle) dans le registre interne Ri.

Tous les vecteurs dont les générateurs d'accès sont linéaires [Jégou 86c] sont accédés par postincrémentation à travers des descripteurs. Les calculs d'adresses sur les vecteurs dont les générateurs d'accès ne sont pas linéaires sont explicites (par l'unité de calcul entier).

Cette boucle contient 22 instructions élémentaires, 2 opérations flottantes, 7 accès à la mémoire, 6 accès à la mémoire locale, 6 instructions à destination de l'unité de calculs entiers, et une instruction sur l'unité de séquencement. Pour atteindre une vitesse de M Mflops sur cette boucle, les débits exigés des unités fonctionnelles sont

↦ 11*M Mips du séquenceur (nombre d'instructions séquencées),
↦ M/2 Mflops de l'additionneur flottant,

↦ M/2 Mflops du multiplieur flottant,
↦ 7*M/2 Maccès sur la mémoire principale,
↦ 3*M Mops de l'unité de calcul entiers,
↦ 3*M Maccès à la mémoire locale.

Une deuxième limite de puissance de cette boucle provient des circuits de dépendance sur les unités fonctionnelles. Cette limite provient du fait que chaque unité fonctionnelle exécute ses instruction dans l'ordre du séquencement global. Lorsqu'une instruction d'une certaine unité fonctionnelle est appliquée à un opérande calculé à partir de valeurs produites par cette même unité fonctionnelle, le délai qui doit s'écouler entre l'exécution de l'instruction productrice et l'exécution de l'instruction consommatrice est supérieur (ou égal) au nombre d'étages de pipelines traversés par les paramètres du calcul. C'est le cas de l'accès aux éléments de vecteurs dont le générateur d'accès est non linéaire et pour lesquels une première instruction de l'unité mémoire lit la valeur d'indices, une deuxième instruction de cette même unité provoquant l'accès à une adresse issue de calculs sur la valeur produite par cette première instruction. Par exemple, pour accéder à l'élément b(c(i)), il faut totaliser la durée de la lecture (postincrémentée) de c(i), la multiplication et l'addition sur l'unité Al. Dans l'exemple traité, en considérant que la mémoire et l'unité de calcul sur les entiers exécutent une instruction par cycle et que le nombre d'étages de pipeline de ces unités soit 8 pour la mémoire et 2 pour l'unité de calcul entière, le délai minimum entre l'exécution de l'instruction de lecture de c(i) et l'instruction de lecture de b(c(i)) est de 12 cycles (8+2+2). Or, en datant au plus tôt les instructions de chaque unité fonctionnelle, l'instruction d'écriture du résultat en mémoire est exécutée 16 cycles après l'exécution de la première instruction de lecture de la même itération. L'écriture effective n'est réalisée que 28 cycles après la première lecture (phénomène de repliage automatique des boucle permis par le doublage des écritures par les lectures). Le débit potentiel de la mémoire est donc proche de la moitié du débit maximum (7 accès en 16 cycles). On voit ici l'importance d'unités de calculs d'adresses rapides dans les cas d'indirections. Une durée de 8 cycles pour les accès à la mémoire peut sembler élevée, mais elle nous semble réaliste en présence de mémoires entrelacées. En fixant la durée du cycle à 50 ns, une vitesse d'exécution de 2.5 Mflops peut être espéré bien qu'aucune optimisation

n'ait été effectuée dans la traduction de la boucle (les instructions sont entrelacées par une technique systématique de génération de code pour des FIFOs que nous décrivons en §8). Cette puissance est proche de celle obtenue sur un code optimisé pour le Cray 1 et dont le cycle d'horloge est 4 fois plus rapide. Un autre point de comparaison possible est de calculer la longueur des pipelines qui permettent d'atteindre une puissance de un Megaflop. Dans l'exemple traité, il suffit que la somme lM+ladd+lmul des longueurs de pipeline lM pour la lecture mémoire, ladd pour l'addition entière et lmul pour la multiplication entière soit inférieure ou égale à 36, ce qui donne, par exemple, 20 pour lM, 8 pour ladd et lmul.

Ml	lire, n	dS
Sq	initBS, Ri, Ml	
Ml	lire, Aa	Alg
M	lpic, Dl	Ald
Ml	lire, Ab	Alg
M	lpic, Dc	Ald
Al	+, Ml, M	Ml
Al	+, Ml, M	aM
Ml	écrire, x, Al	
Ml	lire, x,	aM
M	lpic, Dv	*fg
M	lire, Al	*fd
M	lire, Ml	+fg
*f	*, M, M	+fd
Ml	lire, x	aM
+f	+, M, *f	dM
M	écrire, Ml, +f	
Sq	itère,Ri	

figure 10

En détectant que les raisons d'accès aux vecteurs a et b sont l'unité et que l'élément a(l(i)) est accédé deux fois dans l'instruction, le code de la figure 10 peut être produit. Ce nouveau codage permet de ramener la durée d'exécution de chaque itération sur la mémoire à 13 cycles. Par

rapport à la solution non optimale, seuls trois cycles ont pu être gagnés bien que le processus de compilation utilisé soit plus complexe. Une autre optimisation possible consiste à utiliser des instructions d'adressage basé que nous avons défini en §4. Dans l'exemple traité, cette solution permettrait de gagner au mieux un ou deux cycles. Les processus d'optimisation complexes ne sont donc pas nécessaires pour atteindre des performances honnêtes sur un DSPA, objectif que nous avions fixé pour cette étude. Ici les limites de performances sont dues à l'ordre de séquencement strict que nous avons adopté sur les unités fonctionnelles. L'introduction d'un mécanisme complexe de recherche des instructions exécutables parmis les instructions de la mémoire d'instruction d'une unité fonctionnelle ne nous semble pas souhaitable, d'une part à cause de sa complexité, d'autre part parce qu'elle ne permet pas l'utilisation de vraies FIFOs. Une solution possible qui respecte totalement le principe du DSPA consiste à définir deux unités fonctionnelles du même type. Ces deux unités fonctionnelles possèdent alors leurs propres FIFOs de données et d'instructions. Mais, elles peuvent se partager les mêmes ressources physiques de séquencement, de décodage, et de traitement. Les rebouclages sont alors éliminés en faisant en sorte que les deux unités fonctionnelles interviennent dans le calcul. Par exemple, dans l'algorithme précédent, une première unité fonctionnelle mémoire serait chargée de lire les vecteurs d'indices, l'autre unité fonctionnelle mémoire étant chargée de lire les données. Mais, la mise en oeuvre d'une telle séparation peut provoquer des interférences avec le système de détection des aléas sur la mémoire. Nous verrons par la suite que l'introduction d'instructions vectorielles dans le DSPA permet d'éviter une telle duplication des unités fonctionnelles dans la grande majorité des cas.

8 génération de code pour le DSPA

Un suite d'instructions de DSPA doit respecter la contrainte suivante : si deux instructions I_1 et I_2 d'une même unité fonctionnelle codées dans cet ordre ont le même destinataire (même entrée logique de la même unité fonctionnelle), les deux instructions consommatrices C_1 et C_2 correspondantes doivent être codées dans ce même ordre. Au premier abord, la technique de génération de code à mettre en oeuvre pour le DSPA ne semble pas évidente. Nous rappelons que pour qu'une telle machine soit

efficace, il ne suffit pas de prouver qu'il existe du code efficace, encore faut-il pouvoir le produire simplement. La technique que nous avons défini est très simple. Elle consiste à développer les arbres d'expressions de manière classique. Par exemple, l'instruction

```
a := b + c*d ;
```

est développée par l'arbre de la figure 11. Chaque noeud de l'arbre

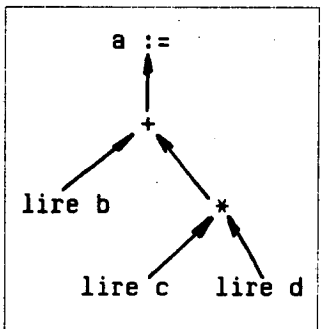

figure 11

contient une instruction d'unité fonctionnelle. Chaque arc correspond à un point de connexion entre deux unités fonctionnelles. La désignation des FIFOs d'entrée de chaque instruction est déduite de l'origine de l'arc correspondant, le désignation des entrées logiques destinataires des résultats étant déduite de la destination de l'arc. Les données progressent toujours des feuilles vers la racine (les feuilles sont en bas et la racine en haut). Cette phase de construction d'arbres d'expressions est suffisemment classique pour ne pas être développée ici. Chaque noeud de l'arbre est situé à une certaine distance de la racine (nombre d'arcs à parcourir pour atteindre la racine). Les instructions situées à la même distance de la racine appartiennent à une même *strate* d'instructions (figure 12). Pour produire un code de DSPA, il suffit d'ordonner ces instructions strate par strate en commençant par la strate la plus éloignée et en prenant les instructions de gauche à droite sur chaque strate. Le résultat de la génération de l'exemple précédent est alors celui de la figure 13. Cette technique très simple permet la production de code pour une machine à une FIFO (la désignation des producteurs et des consommateurs est implicite). Les productions de code que nous présentons par la suite

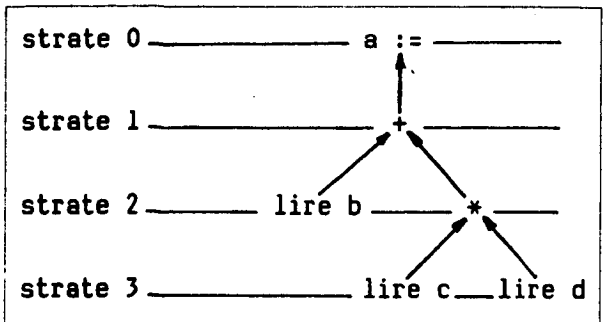

figure 12

```
lire c
lire d
lire b
*
+
a :=
```

figure 13

sont toutes basées sur cette technique de base. Elles consistent à produire du code pour les machines suivantes :

(1)↦ **les machines à une FIFO.** Sur ces machines, l'ordre de production des instructions est strict.

(2)↦ **les machine à une FIFO d'entrée de donnée par unité fonctionnelle.** Dans ce cas, à chaque instruction est associé le numéro de l'unité fonctionnelle destinataire. Par rapport au code à une seule FIFO, deux instructions successives qui ne s'adressent pas à la même unité fonctionnelle et qui n'indiquent pas le même destinataire sont indépendantes et peuvent donc être permutées ou exécutées en parallèle. Cependant, sur une telle machine, les unités fonctionnelles ne sont pas indépendantes car elles se partagent les FIFOs en écriture. Le code des machines à une seule FIFO s'exécute correctement sur une telle machine à condition de lui ajouter l'identification du destinataire.

(3)↦ **les machines à une FIFO par entrée logique d'unité fonctionnelle.** Par

rapport au cas précédent, la distinction des entrées logique permet de relever plus d'indépendance dans les suites d'instructions. Les deux branches de production vers les deux entrées d'une opération binaire peuvent être échangées dans l'arbre de l'expression. On peut alors favoriser la branche la plus critique. Les instructions de type (1) et (2) sont exécutables sur ces machines. Cependant, les unités fonctionnelles partagent les FIFOs en écriture et ne sont donc pas indépendantes.

(4)↦ **les machines à une FIFO par sortie d'unité fonctionnelle.** Sur ces machines, chaque instruction fait référence aux unités fonctionnelles qui produisent les opérandes, la destination étant implicitement la FIFO de sortie de l'unité fonctionnelle. Dan ce cas, deux instructions successives qui ne s'adressent pas à la même unité fonctionnelle et qui ne font pas référence aux mêmes producteurs sont indépendantes et peuvent être échangées. Le code de type (1) est exécutable sur une telle machine. Les unités fonctionnelles se partagent les FIFOs en lectures et ne sont donc pas indépendantes. Comme nous le verrons par la suite, on peut dans ce cas associer un poids aux arcs de sortie des unités fonctionnelles pour favoriser les unités fonctionnelles critiques.

(5)↦ **les machines qui possèdent une FIFO par connexion sortie d'unité fonctionnelle ↦ entrée d'unité fonctionnelle (cas du DSPA).** Ces machines peuvent exécuter les codes de toutes les machines précédentes. Dans ce cas, deux instructions successives sont dépendantes uniquement si elles s'adressent à la même unité fonctionnelle et qu'elles désignent soit une même entrée d'unité fonctionnelle destinatrice, soit un même producteur sur une même entrée logique.

La séquence produite pour une machine à une seule FIFO peut être directement transformée en code pour une machine à plusieurs FIFOs. Il suffit pour celà de désigner explicitement les FIFOs dans les instructions sans modifier leur ordre. Lorsque l'accès aux données provoque des calculs d'adresses, les instructions correspondantes sont également associées à l'arbre. Par exemple, l'instruction du corps de la boucle de multiplication de matrice creuse par un vecteur plein précédent peut être développé sous la forme de la figure 14. Nous ne faisons pas apparaître les noms d'unités fonctionnelles ni leurs entrées pour éviter d'allourdir cette figure (arbre de générarion pour une machine à une seule FIFO). Dans

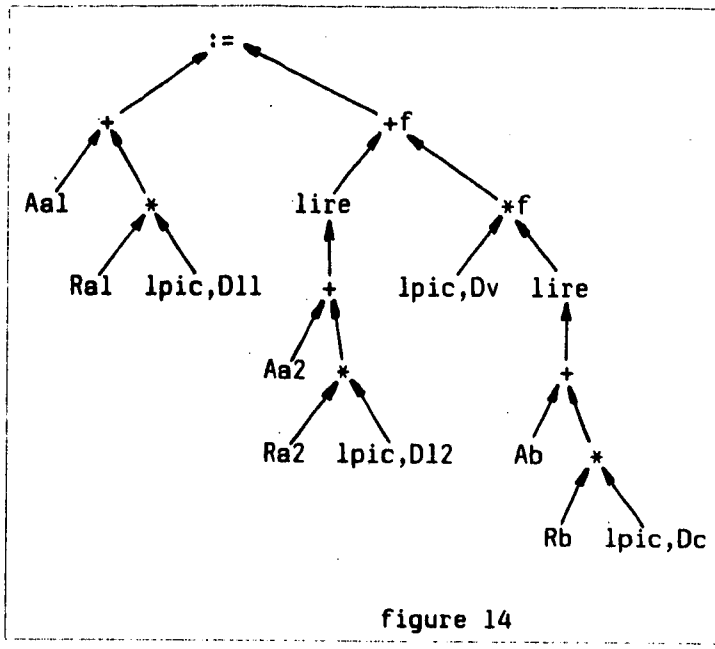

figure 14

la pratique, la construction de l'arbre et la construction des instructions peuvent être faits simultanément. En appliquant la technique de génération par strate, on obtient la séquence de code dit non optimal du paragraphe §7.

Les séquencements des instructions des unités fonctionnelles étant indépendants sur une machine à plusieurs FIFOs comme le DSPA, il n'est pas nécessaire de produire une suite de codes globale. Il suffit en effet de produire une suite par unité fonctionnelle. Cette séparation permet de simplifier le processus de génération dans le cas où plusieurs instructions d'unité fonctionnelle peuvent être codées dans un mot de la mémoire de programme (séquencement de plusieurs instructions en parallèle). En appliquant cette possibilité à l'arbre précédent, on obtient les 7 séquences de code suivantes.

● de la mémoire

M	lpic, Dc	Ald
M	lpic, D12	Ald
M	lpic, D11	Ald

```
M  |   lpic, Dv      | *fg
M  |   lire, Al      | *fd
M  |   lire, Al      | +fg
M  | écrire, Al, +f  |
```

● de la mémoire locale

```
Ml |   lire, Rb      | Alg
Ml |   lire, Ra2     | Alg
Ml |   lire, Ab      | Alg
Ml |   lire, Aa2     | Alg
Ml |   lire, Ral     | Alg
Ml |   lire, Aal     | Alg
```

● de l'unité de calcul entière

```
Al |      *, Ml,  M | Ald
Al |      *, Ml,  M | Ald
Al |      +, Ml, Al | aM
Al |      +, Ml, Al | aM
Al |      *, Ml,  M | Ald
Al |      +, Ml, Al | aM
```

● du multiplieur flottant

```
*f |      *,  M,  M | +fd
```

● de l'additionneur flottant

```
+f |      +,  M, *f | dM
```

Ces 7 séquences de code peuvent être fusionnés librement, seul l'ordre des instructions pour une même unité fonctionnelle étant significatif (à chaque FIFO sont associées une seule unité fonctionnelle productrice et une seule unité fonctionnelle consommatrice). Le fait que l'instruction qui produit une donnée puisse appraître avant l'instruction qui consomme cette donnée ne pose aucun problème, la synchronisation étant garantie par l'utilisation de FIFOs d'échange des données et la présence des FIFOs d'instructions.

Considérons un DSPA dont chaque mot de la mémoire de programme possède un champs par unité fonctionnelle (type microprogrammes). Le corps de la boucle de multiplication matrice creuse par vecteur plein peut alors être codée de la manière suivante en juxtaposant les 7 séquences d'instructions des 7 unités fonctionnelles. Il suffit de respecter les points de rupture de séquence (instructions de saut et instructions vers lesquelles un saut peut être fait) pour réaliser correctement un tel repliage des instructions (figure 15). Pour des problèmes d'espace, nous ne faisons pas apparaître

Mémoire		Mémoire locale		Alu entière		Multiplieur		Additionneur	
lpic,Dc	Ald	lire,Rb	Alg	*,Ml,M	Ald	*,M,M	+fd	+,M,M	dM
lpic,Dl2	Ald	lire,Ra2	Alg	*,ML,M	Ald				
lpic,Dl1	Ald	lire,Ab	Alg	+,Ml,Al	aM				
lpic,Dv	*fg	lire,Aa2	Alg	+,Ml,Al	aM				
lire,Al	*fd	lire,Ral	Alg	*,Ml,M	Ald				
lire,Al	+fg	lire,Aal	Alg	+,Ml,Al	aM				
écrire,Al,+f									

figure 15

les instructions du séquenceur. L'initialisation de la boucle par l'instruction initBS est codée dans le mot qui précède la première instruction de la boucle. L'instruction itère est codée dans la dernière instruction de la boucle. Dans un tel repliage, de nombreux champs d'unités fonctionnelles peuvent rester vides. Des repliages intermédiaires peuvent être réalisés, par exemple, en allouant un champs aux unités fonctionnelles les plus requises et en fusionnant les séquences d'instructions des unités fonctionnelles qui se partagent un même champs. Une dernière solution encore plus compacte consiste à prévoir par exemple des mots de 4 champs de même taille et non spécialisés. Les instructions d'une unité fonctionnelle peuvent être placée dans n'importe quel champs. Un algorithme simple permet alors de fusionner les suites d'instructions de telle façon qu'un même mot ne contiennent pas deux instructions à destination d'une même unité fonctionnelle (ce qui permet de garantir le séquencement de 4 instructions par cycle tout en limitant le débit en

écriture des FIFOs d'instructions à 1 écriture par cycle) en donnant la priorité à la séquence la plus longue.

La technique de génération de code DSPA que nous avons exposé est très simple et donne de très bons résultat pour chaque instruction et lorsque les longueurs des pipelines sont proches. Elle peut être améliorée par fusion d'instructions successives d'un programme et en donnant une priorité de génération aux branches les plus longues des arbres.

a) fusion d'arbres

La première idée consiste à concaténer les instructions de chaque unité fonctionnelle entre chaque point de rupture de séquence. Cette concaténation permet d'équilibrer la charge des unités fonctionnelles, et donc un meilleur compactage du code. Mais l'ordre des instructions de chaque unité fonctionnelle n'est pas modifié. Cependant, dans les cas de rebouclage trop serrés sur une même unité fonctionnelle (fréquents dans les doubles indexations comme nous l'avons vu dans l'algorithme de multiplication de matrice creuse par un vecteur plein), il faut chercher à éloigner au maximum les instructions de production des instructions de consommation. Pour celà, il suffit de fusionner les instructions correspondant à plusieurs arbres successifs à condition qu'aucun point de rupture de séquence ne les sépare. Cette fusion ne peut pas être réalisée directement à partir du code produit car il faut respecter les dépendances des données. Par contre, rien n'interdit de juxtaposer les deux arbres comme dans le cas des deux instructions indépendantes suivantes :

```
a := b + c ;
d := e + f*g ;
```

les deux arbres correspondants peuvent être juxtaposés comme dans la figure 16. Dans ce cas, chaque strate contient des instructions en provenance des deux arbres. Le code produit (figure 17) est un mélange d'instructions en provenance de ces deux arbres. C'est un code pour une machine à une seule FIFO, donc indépendant de la structuration réelle du processeur. Cette fusion a pour effet d'éloigner les instructions de production de données des instructions de consommation. Rappelons que cette technique n'a d'effet réel que lorsque des rebouclages de données se produisent sur une

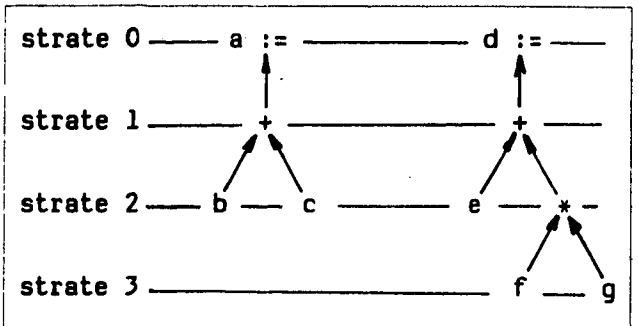

figure 16

```
lire f
lire g
lire b
lire c
lire e
*
+
+
a :=
d :=
```

figure 17

même unité fonctionnelle (ce qui n'est pas le cas de cet exemple). Comme les instructions d'accès à la mémoire (ou à la mémoire locale) ne sont plus exécutées dans le même ordre, cette juxtaposition des arbres doit tenir compte des dépendances possibles. Lorsqu'une telle dépendance existe, il suffit alors de faire en sorte que l'instruction d'écriture se trouve sur une strate de distance supérieure ou égale à la strate où se trouve la lecture. Par exemple, soit la séquence

```
a := b + c ;
d := a + f*g ;
```

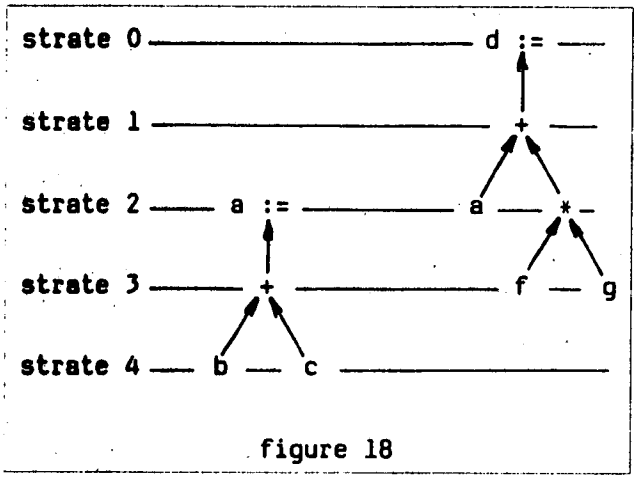

figure 18

Les arbres de ces deux instructions peuvent être juxtaposés comme dans la figure 18 pour faire en sorte que l'instruction d'écriture de a soit exécutée avant l'instruction de lecture de a. La séquence de la figure 19 est alors produite. Cette façon d'aligner les strates permet une fusion

```
lire b
lire c
+
lire f
lire g
a :=
lire a
*
+
d :=
```

figure 19

efficace. C'est de cette manière que nous avons obtenu le code dit optimal du paragraphe §8, figure 10 où un premier arbre défini le calcul d'adresse de a(1(i)) rangé dans la mémoire locale, et un deuxième arbre utilise cette adresse (figure 20). Cette technique de juxtaposition d'arbres a cependant quelques limites. En effet, un éloignement des instructions de production

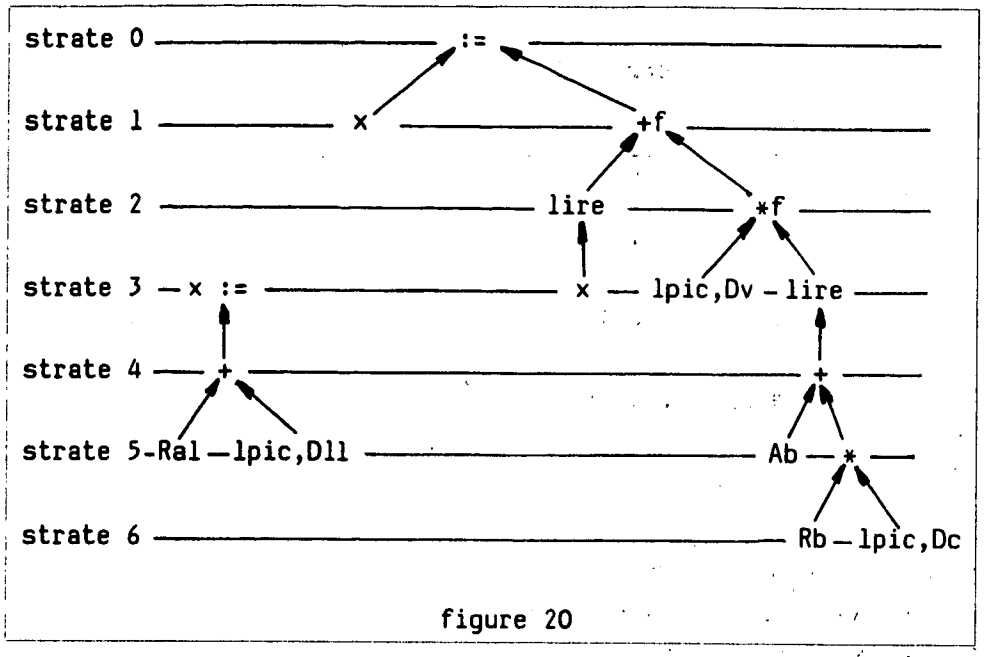

figure 20

par rapport aux instructions de consommation se traduit par l'insertion d'instructions à destination de la même unité fonctionnelle, effet très visible sur une machine à une seule FIFO. Le cas limite qui peut se produire est que le séquencement d'une unité fonctionnelle productrice soit suspendu pour cause de saturation des FIFOs destinatrices, cet arrêt de décodage entraînant la saturation de sa FIFO d'instructions et donc du séquencement global. Il y a blocage si l'instruction de consommation n'a pas été séquencée. Par exemple, en supposant que toutes les FIFOs (instructions et données) soient de longueur maximum 8, un tel blocage peut se produire lorsqu'une certaine unité fonctionnelle a exécuté 8 instructions de même destination, qu'elle a reçu 8 autres instructions du séquenceur alors que l'instruction de consommation correspondant à la première valeur produite n'a pas été séquencée. Ce cas limite est fortement improbable, mais doit être traité. Nous verrons en §9 que cette limite peut être atteinte plus rapidement dans le cas des instructions vectorielles.

b) l'équilibrage du pipeline

L'ordre de production de code que nous avons défini ne tient pas compte des délais dûs aux pipelines des unités fonctionnelles ni des longueurs des enchaînements des unités fonctionnelle (c'est un code pour une machine à une seule FIFO). Il paraît en effet logique que le fait de donner une certaine priorité aux instructions de la branche la plus longue (en étages de pipeline) ne peut qu'améliorer le comportement du DSPA. Dans la technique exposée précédemment, les arcs liant les instructions sont de même longueur unité, et la distance d'un noeud à la racine qui permet de déterminer sa strate est obtenue en cumulant le nombre d'arcs pour atteindre la racine. Mais, pour que les contraintes d'ordre sur les instructions productrices et consommatrices soient respectées, il n'est pas nécessaire que tous les arcs soient de même longueur : il suffit en effet que les arcs correspondant à une même FIFO (qui relient une sortie d'unité fonctionnelle à une même entrée logique d'une même unité fonctionnelle) soient de même longueur. Un équilibrage simple consiste à affecter aux arcs de sortie des unités fonctionnelles une longueur correspondant au nombre d'étage moyen du pipeline de l'unité fonctionnelle. En prenant des longueurs d'arcs de 8 pour les lectures mémoire, 1 pour la mémoire locale, 2 pour les opérations entières et 4 pour les opérations flottantes, l'algorithme de multiplication matrice creuse par vecteur plein peut être équilibré par l'arbre de la figure 21. Une génération par strate donne le résultat de la figure 22.

Cette technique permet de produire du code pour une machine à une FIFO par sortie d'unité fonctionnelle, qui englobe donc le cas du DSPA. En principe, cette nouvelle séquence devrait être plus rapide que la précédente car l'ordre d'émission des requêtes dépend des délais induits par les pipelines. Mais, dans le DSPA, nous avons introduit des mécanismes de doublement des lectures sur les écritures mémoire pour essayer de cacher ces effets de pipeline à l'exécution. Ces deux optimisations peuvent interférer. C'est ce qui se passe dans cet exemple qui est en réalité moins rapide (1 cycle) que le précédent. Cette dégradation provient du fait que nous avons favorisé la branche de calcul des données qui est la plus longue par rapport à la branche de calcul d'adresse d'écriture. Comme seule cette dernière branche provoque un rebouclage systématique sur une unité fonctionnelle, elle devrait être prioritaire. Ce phénomène prouve une fois de plus, que l'objectif fixé qui était de rendre le processeur le moins sensible possible à la qualité du code produit par des compilateurs

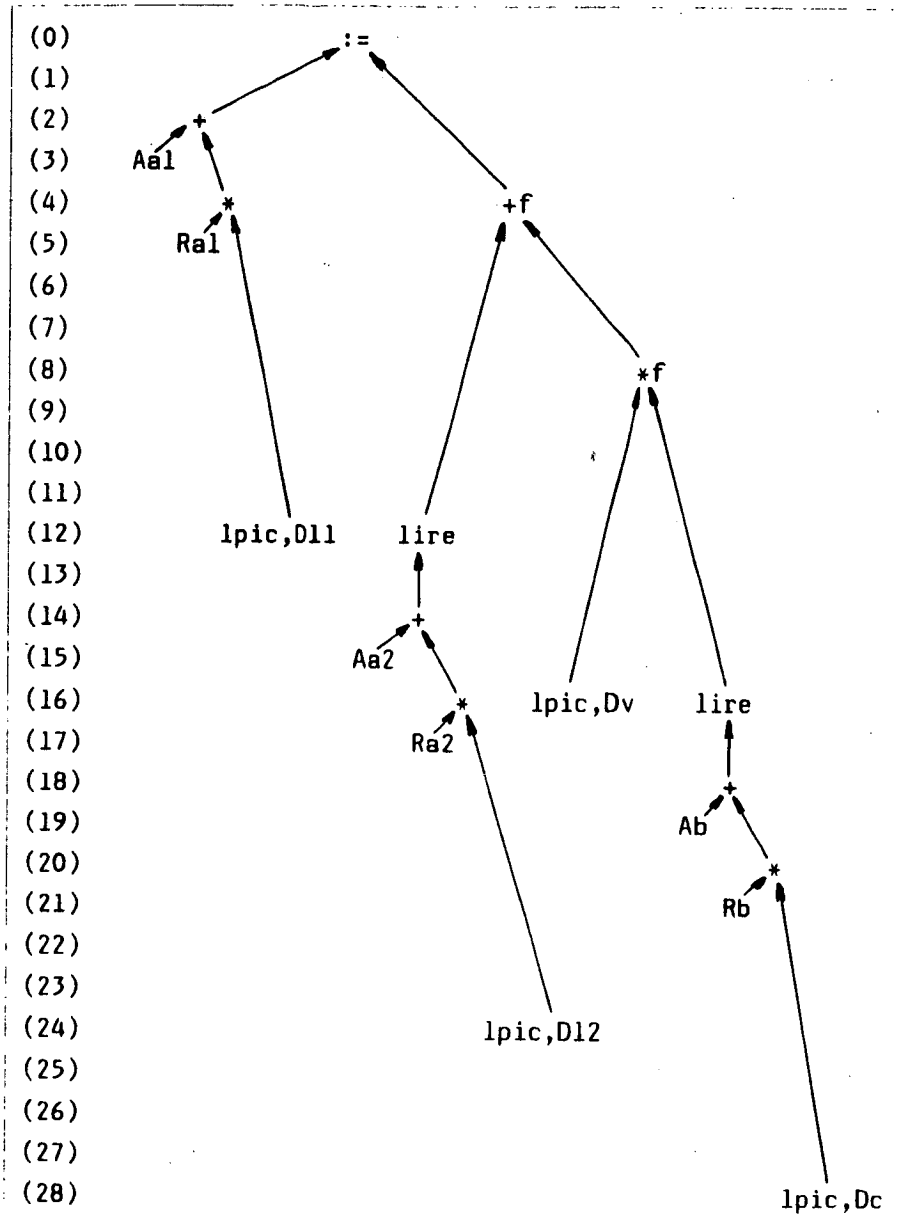
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)
(10)
(11)
(12)
(13)
(14)
(15)
(16)
(17)
(18)
(19)
(20)
(21)
(22)
(23)
(24)
(25)
(26)
(27)
(28)
:=
+
Aa1
*
Ra1
lpic,D11
+f
lire
+
Aa2
*
Ra2
lpic,D12
*f
lpic,Dv
lire
+
Ab
*
Rb
lpic,Dc

figure 21

Mémoire		Mémoire locale		Alu entière		Multiplieur		Additionneur	
lpic,Dc	Ald	lire,Rb	Alg	*,M1,M	Ald	*,M,M	+fd	+,M,M	dM
lpic,D12	Ald	lire,Ab	Alg	+,M1,A1	aM				
lpic,Dv	*fg	lire,Ra2	Alg	*,ML,M	Ald				
lire,A1	*fd	lire,Aa2	Alg	+,M1,A1	aM				
lpic,D11	Ald	lire,Ra1	Alg	*,M1,M	Ald				
lire,A1	+fg	lire,Aa1	Alg	+,M1,A1	aM				
écrire,A1,+f									

figure 22

est atteint. Il aurait fallu n'augmenter que le poids des arcs correspondant aux calculs des adresses. Dans le cas général, l'optimisation présentée permet de gagner quelques cycles et devrait être adoptée. Cet exemple que nous avons volontairement choisi montre à la fois les possibilités offertes par cet équilibrage et ses limites.

Une solution plus efficace consiste à prendre en compte l'ensemble des FIFOs du DSPA et de donner un poids (priorité) plus élevé aux arcs correspondant aux FIFOs placées sur les chemins critiques. Un chemin qui part d'une certaine unité fonctionnelle et qui aboutit à la même unité fonctionnelle est considéré comme critique. Pour éloigner l'instruction de production (départ d'un chemin critique) de l'instruction de consommation, il suffit d'allonger ce chemin en allongeant un ou plusieurs des arcs composant ce chemin. Lorsque la longeur d'un arc est modifiée, il faut également affecter la même longueur à tous les arcs correspondant à la même FIFO (règle de production de code sur les FIFOs). Cette technique est légèrement plus complexe que la précédente, mais elle a l'avantage de ne jamais dégrader les performances par rapport à la génération de code de base pour une seule FIFO. Dans l'exemple de la multiplication d'une matrice creuse par un vecteur plein, le code produit donne les mêmes performances que le code a une seule FIFO.

Une autre optimisation du même type et qui n'est pas incompatible avec les précédentes consiste à considérer que deux branches qui aboutissent à

un même noeud peuvent être commutées. En effet, rien n'impose que les instructions qui produisent sur une entrée logique gauche soient produites avant les instructions qui produisent sur les entrées logiques droites. Il suffit alors de faire en sorte que, pour chaque instruction, la branche la plus prioritaire soit placée à gauche (la production des instructions d'une strate se faisant de gauche à droite). Comme l'optimisation précédente, cette réorganisation permet de gagner quelques cycles dans le cas général. Une optimisation du même type consiste à placer à gauche les branches qui provoquent des rebouclages sur des unités fonctionnelles. Enfin, de nombreuses manipulations d'arbres imaginables sont possibles. Les seules règles à respecter sont que, les arcs qui correspondent à une même FIFO soient de même longueur, la structuration en strates soit conservée et corresponde aux longueurs des arcs, et enfin que la représentation de l'arbre soit planaire (deux arcs ne se croisent pas).

9 les instructions vectorielles

Pour atteindre l'objectif fixé qui était l'exécution rapide de code scalaire, nous avons limité la description du DSPA au fonctionnement scalaire. Mais, dans un cadre plus général, l'introduction d'instructions vectorielles sur un processeur permet de limiter certaines exigences de débits de séquencement.

Sur un processeur DSPA, une instruction vectorielle est séquencée comme une instruction scalaire, mais elle est exécutée plusieurs fois par l'unité fonctionnelle. Pour des raisons pratiques (capacité de mémorisation interne principalement), la longueur des vecteurs traités par les instructions vectorielles de base doit cependant être limité. La compilation des instructions vectorielles entraîne donc un découpage vertical (voir [Jégou 86c]). Dans les exemples traités, nous prenons 8 comme longueur des vecteurs. Pour éviter d'avoir à traiter les vecteurs de longueur inférieure en mode scalaire (traitement complexe, cette longueur n'étant pas toujours connue à la compilation), la longueur courante doit pouvoir être définie dynamiquement. A chaque instruction vectorielle émise par le séquenceur vers chaque unité fonctionnelle doit être associée la valeur courante de la longueur des vecteurs traités. En effet, les séquencements des unité fonctionnelles étant indépendants, cette valeur

peut avoir été changée entre le séquencement de l'instruction et son exécution. La longueur courante des vecteurs fait donc partie des variables d'état du séquenceur. Cette valeur ne peut être modifiée que par des instructions de cette unité.

Dans la FIFO d'instructions de chaque unité fonctionnelle, une longueur LV de vecteur est associée à chaque instruction. Au décodage local, cette valeur n'est pas prise en compte pour le séquencement des instructions scalaires. Par contre, dans les cas des instructions vectorielle, cette valeur indique le nombre de fois que l'instruction doit être répétée. Par exemple, soit l'instruction Hellena

```
a := b + c ;
```

sur des vecteurs a, b et c de longueur n. Par tronçonnage vertical ([Jégou 86c]), cette instruction est traduite par la boucle de la figure 23 sur le DSPA. Les instructions de cet exemple qui commencent par la lettre

```
| Ml | lire,n       |  dS  |
| Sq | initBV,Ml    |      |
| M  | Vlpic,Db     | +fg  |
| M  | Vlpic,Dc     | +fd  |
| +f | V+, M, M     |  dM  |
| M  | Vepic,Da, +f |      |
| Sq | itèreLV      |      |
```

figure 23

V sont vectorielles. La première instruction de cette séquence lit la valeur n en mémoire locale et l'émet vers l'entré logique dS du séquenceur. L'instruction initBV a deux rôles : elle provoque l'initialisation d'un registre interne VL par la valeur reçue sur la FIFO désignée sur l'entrée logique dS (Ml dans l'exemple) ; elle provoque également la mémorisation de la valeur courante du compteur ordinal, valeur utilisée par l'instruction d'itération vectorielle itèreLV. Cette dernière instruction itèreLV provoque le passage en séquence si la valeur courante du registre interne VL est inférieure ou égale à la longueur maximum 8. Dans le cas contraire,

elle provoque un saut à l'adresse mémorisée par l'instruction initBV et décrémente le registre VL de 8. Ces instructions étant exécutées par le séquenceur, on garantit que le registre interne VL est à jour lors du séquencement des instructions vectorielles. La valeur LV associée aux instructions vectorielles de cette boucle est min(8,VL). En fait, dans ce cas, la valeur associée aux instructions vectorielles est toujours 8 sauf pour la dernière itération.

L'existence d'instructions vectorielles présente plusieurs intérêts.

↦ Le débit de séquencement des instructions peut être limité.

↦ Dans de nombreux cas d'adressages linéaires, le comportement de la mémoire peut être guaranti, surtout dans les adressages linéaires de raison impaire d'une mémoire composée d'une puissance de 2 de bancs entrelacés.

↦ Certains comportements des pipelines sont améliorés, principalement lorsque des rebouclages existent sur des unités fonctionnelles. C'est par exemple le cas de l'instruction vectorielle suivante.

```
a := b + c + d ;
```

Exécutée en mode scalaire, le résultat de l'addition de c à d est émis sur l'entré droite de l'additionneur et consommé par l'instruction suivante, ce qui provoque un vidage du pipeline de l'additionneur. Exécutée en vectoriel, ce résultat est demandé au plus tôt LV (longueur du vecteur) cycles d'additionneur plus tard. Il suffit alors que la longueur courante des vecteurs soit supérieure au nombre d'étages du pipeline pour obtenir un débit optimal d'exécution.

La définition des instructions vectorielles doit être généralisée aux opérations scalaires-vecteurs pour éviter d'avoir à dupliquer les scalaires dans les FIFOs d'entrée des opérateurs. Pour celà, on peut, prévoir 4 codes différents pour chaque instruction pour les cas ou les deux opérandes sont scalaires, les deux opérandes sont vectoriels, l'opérande gauche est scalaire alors que l'opérande droit est vectoriel, et enfin le cas où l'opérande gauche est vectoriel alors que l'opérande droit est scalaire. Une autre solution consiste à définir un seul code pour l'instruction et à

indiquer pour chaque opérande s'il est scalaire ou vectoriel (1 bit par opérande). Pour exécuter une opération scalaire-vecteur, le scalaire est extrait de sa FIFO et conservé dans l'opérateur pendant le séquencement de l'instruction vectorielle. En général, lorsque au moins 1 opérande est vectoriel, l'instruction est vectorielle et le résultat est également vectoriel, mais ceci dépend de l'instruction exécutée. Dans tous les cas, l'utilisation des FIFOs d'échange évite d'avoir à spécifier la catégorie du résultat.

La signification de chaque instruction doit être soigneusement définie dans les cas scalaires et dans les cas vectoriels.

→ Sur le séquenceur, il n'y a pas d'instructions vectorielles.

→ Sur les opérateurs flottants, l'instruction scalaire est répétée.

→ Sur la mémoire principale, il faut distinguer les instructions qui prennent leur adresses sur les entrées logiques des instructions d'adressage sur les descripteurs. Les instructions de lecture vectorielle n'ayant qu'un seul opérande (les adresses), cet opérande est nécessairement vectoriel. Une instruction du type *lire* qui fait référence à l'entrée logique d'adresses provoque LV lectures sur les LV premières adresses de la FIFO désignée. Cette instruction permet de réaliser les lectures mémoires sur des vecteurs d'adresses et par conséquence de traduire des accès vectoriels de type *gather*. La présence d'adressages non linéaires n'est donc pas un frein à la production de code vectoriel sur un DSPA. Pour le cas de lectures avec adressage basé, la base est considérée comme un scalaire qui est ajouté au vecteur des déplacements extraits de l'une des FIFOs d'adresses.
Les instructions de lecture vectorielle avec postincrémentation d'un descripteur provoquent des accès aux adresses de la forme A+I×R, I variant de 0 à LV-1 et la mise à jour de l'adresse de base par A+LV×R. Les adresses peuvent être produites par incrémentations successives, ou en utilisant un opérateur spécialisé. Les instructions de dépilement vectoriels sont traduites de manière similaire.

Les instructions d'écriture mémoire ont deux opérandes. Mais les cas d'écritures dont l'adresse est un scalaire et la donnée un vecteur n'ont

pas de signification véritable. Par contre, les cas où l'adresse est vectorielle et la donnée scalaire correspondent à l'écriture d'une même valeur dans un vecteur.

Enfin, les instructions d'initialisation de descripteurs ne peuvent pas être vectorielles.

↦ La mémoire locale. Contrairement à la mémoire principale, nous n'avons pas prévu d'entrée logique de calcul d'adresses sur cette unité fonctionnelle, elle est utilisée comme un bloc de registres. Pour les instructions vectorielles, il faut cependant être capable de désigner un ensemble de mots de cette mémoire. La solution la plus simple consiste à considérer qu'un accès vectoriel à la mémoire locale provoque l'accès à LV mots contigüs. Comme nous le verrons par la suite, le DSPA permet l'accès à des opérandes vectoriels par des instructions scalaires et vice-versa. On doit alors être capable d'accéder séquentiellement à une suite de mots contigüs de cette mémoire en scalaire. Ceci pourrait être rendu possible par l'introduction d'une entrée logique d'adresses, ou en considérant une instruction de lecture scalaire qui reçoit une adresse sur l'entrée de données et une instruction d'écriture qui reçoit à la fois la donnée et l'adresse sur la même entrée logique. Rappelons à ce sujet que les entrées logiques des opérateurs correspondent aux opérandes logiques des instructions, et que le fait de les distinguer physiquement permet de garantir que tous les opérandes sont extraits de FIFOs distinctes et peuvent donc être accédés en parallèle. Mais rien dans le principe du DSPA n'interdit l'existence d'instructions qui prennent plusieurs opérandes sur une même entrée logique. Cette distinction est par contre nécessaire pour les instructions vectorielles (pour éviter des difficultées lorsque les deux opérandes sont produits par une même unité fonctionnelle). Cependant, comme nous l'avons fait pour la mémoire, nous introduisons un ensemble de descripteurs dans l'unité fonctionnelle mémoire locale ainsi que deux instructions de lecture et d'écriture scalaires avec postincrémentation (de 1) sur ces descripteurs. De cette façon, l'accès à un vecteur rangé dans la mémoire locale peut se faire efficacement dans une boucle d'instructions scalaires. Pour être conforme à l'adressage absolu codé dans les instructions, nous nous contentons d'initialisations de ces descripteurs par des adresses absolues codées dans ces instructions d'initialisation.

↦ L'unité de calcul d'entier. Cette unité a le même comportement que les opérateurs flottants pour les instructions dont les opérandes sont extraits des FIFOs d'entrées. Les instructions vectorielles de postincrémentation de descripteurs produisent des séquences linéaires de valeurs. Par contre, les instructions d'initialisation de descripteurs n'ont pas de signification vectorielle.

Ces instructions vectorielles permettent une traduction directe des instructions vectorielles et des instructions **pour_tout** du langage Hellena. Comme pour les instructions scalaires, les lectures peuvent doubler de façon contrôlée les écritures sur la mémoire. En fait, la notion d'instruction vectorielle est prise en compte au niveau du séquencement interne de chaque unité fonctionnelle et traduite en une suite d'instructions scalaires. Une instruction vectorielle de longueur LV sur la mémoire provoque l'émission de LV requêtes vers la mémoire par l'unité fonctionnelle. Les dépendances sont donc calculées sur des vecteurs, ce qui interdit la traduction des instruction **pour_tout** avec spécification de dépendance séquentielle en instruction vectorielle. Par contre, les opérandes vectoriels étant rangés séquentiellement dans les FIFOs d'échange, il n'est pas interdit de consommer un vecteur par une suite d'instructions scalaires, ni de consommer une suite de scalaires par une instruction vectorielle. Nous reprenons l'exemple de la multiplication de matrice creuse par un vecteur plein.

```
pour_tout i sequentiellement dans [1,n] :
   a(l(i)) := a(l(i)) + v(i)*b(c(i)) ;
fin_pour_tout ;
```

Dans cette instruction, seuls les accès au vecteur a doivent être faits séquentiellement pour respecter les dépendances. Les autres accès ainsi que les opérations de calcul des adresses et des valeurs peuvent être exécutées en mode vectoriel. Cet algorithme est codé par deux boucles emboîtées (figure 24). La boucle externe est initialisé en (2) et gère le registre interne VL du séquenceur. La boucle interne permet de traiter les accès au vecteur a en scalaire pour respecter les dépendances séquentielles d'accès. Le compteur R1 de cette boucle est initialisé en (13) à partir de la valeur courante du registre interne LV par une instruction inititLV,R1. Le rebouclage est géré par l'instruction d'itération en (18). Dans la

Ml	lire,n	dS	(1)
Sq	initBV,Ml		(2)
M	Vlpic,Dl	Alg	(3)
Ml	lire,Aa	Ald	(4)
Al	VS+,M,Ml	Ml	(5)
Ml	Vécrire,Ad,Al		(6)
M	Vlpic,Dc	aM	(7)
M	Vlpic,Dv	*fg	(8)
M	Vlbase,Db,Ml	*fd	(9)
*f	*, M, M	+fd	(10)
Ml	init,Ad,Dal		(11)
Ml	init,Ad,Da2		(12)
Sq	inititLV,Rl		(13)
Ml	lpic,Dal	aM	(14)
Ml	lpic,Da2	aM	(15)
M	lire,Ml	+fg	(16)
M	écrire,Ml,+f		(17)
Sq	itère,Rl		(18)
Sq	itèreLV		(19)

figure 24

traduction de l'instruction Hellena, les adresses des éléments du vecteur a sont rangés sous forme de vecteur en mémoire locale par l'instruction (6) d'écriture vectorielle (en contigü à partir de l'adresse absolue Ad). Ce vecteur d'adresses est lu à l'intérieur de la boucle interne par les instructions de lecture scalaire (14) et (15) avec postincrémentation (par 1) de descripteurs initialisés en (11) et (12) par l'adresse absolue Ad. Notons que cette extraction d'instructions pour traitement scalaire à l'intérieur d'une boucle vectorielle est rendu possible par l'utilisation de FIFOs d'échange des données. Cette extraction entraîne en effet l'émission d'instructions de consommation avant l'émission de certaines instructions de production. Dans l'exemple précédent, l'instruction (16) de lecture sur a est émise après l'instruction (10) qui consomme la donnée lue.

Nous ne connaissons pas de technique permettant de réaliser automatiquement de telles extractions dans le cas général. Elle est cependant possible dans la plupart des cas de programmes où la dépendance doit être respectée entre une seule instruction d'écriture et une seule instruction de lecture en mémoire, cas le plus fréquent en analyse numérique. Pour les cas où, dans une instruction pour_tout séquentielle, ce sont les mêmes éléments qui sont accédé en partie gauche et en partie droite d'une instruction, et que les dépendances sont séquentielles sur ces seuls éléments, André Seznec a proposé d'utiliser l'instruction lire/écrire sur la mémoire. Cette instruction peut être associée aux modes d'adressage classiques posticrémentés, basés, direct etc.. L'exécution de cette instruction provoque l'exécution d'une requête de lecture immédiatement suivie d'une requête d'écriture à la même adresse sur le banc. Cette interprétation garantit qu'aucune autre requête ne peut s'intercaler entre la lecture et l'écriture sur la même adresse. Lorsqu'elle est exécutée en vectoriel, elle produit le même effet que l'exécution de paires lire-écrire à la même adresse en séquentiel. En utilisant cette instruction, la codage

Ml	lire,n	dS	(1)
Sq	initBV,Ml		(2)
M	Vlpic,Dc	aM	(3)
M	Vlpic,Dv	*fg	(4)
M	Vlbase,Db,M	*fd	(5)
*f	*, M, M	+fd	(6)
+f	+, M, *f	dM	(7)
M	Vlpic,Dl	aM	(8)
M	Vlebase,Da,M,+f	+fg	(9)
Sq	itèreLV		(10)

figure 25

de l'instruction de multiplication de matrice creuse par un vecteur plein peut être entièrement vectoriel (figure 25). L'instruction (9) provoque une lecture/écriture à une adresse basée par le desripteur Da. Cette instruction permet un code plus compact, mais sa production automatique n'est possible que dans un nombre très limité de cas. Elle permet cependant de nombreuses optimisations manuelles.

10 le masquage

Les instructions vectorielles du DSPA permettent la traduction directe des instructions vectorielles et de nombreuses instructions parallèles **pour_tout** du langage vectoriel Hellena. Cependant, les instructions que nous avons introduit ne permettent pas de traiter directement lés conditionelles des instructions **pour_tout**. On pourrait imaginer un passage en séquentiel comme dans les cas de dépendances. Nous préférons introduire une notion de masquage classique sur les architectures SIMD et pipeline.

Comme nous l'avons fait pour la longueur des vecteurs, nous introduisons dans le séquenceur un registre de masquage qui possède un bit par élément de vecteur (8 bits lorsque la longueur des vecteurs est limitée à 8). Ce registre fait partie intégrante des informations d'état du séquenceur. Dans une première approche, nous considérons que ce vecteur de masque est associé à chaque instruction émise dans la FIFO des instructions des unités fonctionnelles. Ce masque n'est pas pris en compte dans l'interprétation des instructions scalaires. En général, lors de l'exécution des instructions vectorielles, les opérations correspondant aux éléments non masqués sont exécutées normalement, les opérations correspondant aux éléments masqués provoquent l'émission d'une donnée fictive vers le destinataire. Ce principe permet une gestion des FIFOs indépendante des valeurs du masque. Sur l'unité fonctionnelle mémoire, toutes les adresses sont calculées, mais seules celles correspondant aux éléments non masqués sont prises en compte par les bancs mémoires (soit que les adresses masquées ne sont pas émises, auquel cas l'unité de réordonnement doit insérer les valeurs fictives, soit qu'elles sont émises, auquel cas ce sont les bancs qui les gèrent). Le comportement du DSPA est alors celui obtenu sans masquage.

Mais, l'utilisation des FIFOs d'échange permet de n'effectuer les calculs que sur les éléments non masqués. Pour celà, il suffit que le séquenceur calcule le nombre d'éléments non masqués des vecteurs à un instant donné. Ce nombre que nous désignons par la suite par LE (longueur effective) définit le nombre d'éléments non masqués échangés dans les FIFOs. Pour les unités fonctionnelles qui extraient leurs opérandes uniquement des FIFOs (les opérateur) et produisent leurs résultats dans les FIFOs, cette valeur LE est associée aux instructions à la place de LV. Le

nombre de cycles nécessaires pour exécuter une instruction vectorielle sur ces unités fonctionnelles est alors égale au nombre d'éléments non masqués des vecteurs. Il y a compression automatique. Notons que, cette valeur peut varier de 0 (tous les éléments sont masqués) à 8 (longueur maximum de vecteur, aucun élément n'étant masqué). Les opérations vectorielles dont l'un des opérandes est scalaires doivent extraire ce scalaire de la FIFO d'entrée même lorsque la longueur effective LE est nulle. Par contre, aucune opération n'a a être exécutée dans ce cas. Certaines unités fonctionnelles gèrent des descripteurs. Ces unités fonctionnelles ont besoin de la longueur réelle LV et du masque. Par exemple, une instruction avec post-incrémentation sur la mémoire calcule des adresses de la forme A+R*i, i variant de 0 à LV-1, l'adresse de base A étant mise à jour par la valeur A+R*LV. Mais, seules les requêtes correspondant aux éléments non masqués sont émises vers la mémoire. Dans ce cas, l'introduction de valeurs fictives dans les FIFOs est éliminée. On peut imaginer des opérateurs qui, à partir des paramètres A, R, LV, et le masque produisent les adresses utiles en un nombre de cycles égal au nombre d'éléments non masqués.

La structure du DSPA nous a donc permis de réaliser une compression automatique des vecteurs masqués, et donc de limiter le nombre d'opérations nécessaires au nombre d'éléments non masqués. Pour que cette compression soit réellement significative, il faut cependant que la longueur maximum des vecteurs soit suffisemment élevée pour éviter les longueurs effectives nulles. Cette compression interdit la fusion de vecteurs dans une FIFO. Mais, il suffit de ranger vectoriellement ces deux vecteurs en mémoire locale pour que la fusion soit réalisée. Par exemple, soit l'expression Hellena

```
pour_tout i dans [1,n] :
   x(i) := si a(i)>eps alors b(i) sinon c(i) fin_si ;
fin_pour_tout ;
```

Cette instruction peut être traduite par la séquence de la figure 26. Supposons qu'aucun élément ne soit masqué au départ. L'instruction vectorielle (5) compare le vecteur a au scalaire eps lu en mémoire locale. Le résultat de la comparaison est émis vers le séquenceur. Ce résultat pourrait être un vecteur. Nous considérons ici que c'est une instruction

Ml	lire,n	dS	(1)
Sq	initBV,Ml		(2)
M	Vlpic,a	+fg	(3)
Ml	lire,eps	+fd	(4)
+f	VS>,M,Ml	cS	(5)
Sq	masquer,+f		(6)
M	Vlpic,b	Ml	(7)
Ml	Vécrire,A		(8)
Sq	invmasque		(9)
M	Vlpic,c	Ml	(10)
Ml	Vécrire,A		(11)
Sq	demasquer		(12)
Ml	Vlire,A	dM	(13)
M	Vepic,x,Ml		(14)
Sq	itèreVL		(15)

figure 26

vectorielle qui produit un scalaire contenant le masque. Ces opérations de réductions (opérations vectorielles qui produisent un résultat scalaire) ne sont pas interdite dans un DSPA. Le registre interne de masque du séquenceur est mis à jour par l'instruction (6). Cette instruction provoque une mise à jour automatique de la valeur LE. L'instruction (9) complémente ce masque et provoque donc une nouvelle mise à jour de la valeur LE. La somme des accès à la mémoire effectués par les instructions (7) et (10) est égale à LV. Il en est de même sur la mémoire locale par les instructions (8) et (12).

L'utilisation d'instructions vectorielles masquées sur des vecteurs de longueur 1 permet par ailleurs d'améliorer le comportement du séquencement du DSPA dans de nombreux cas où les branches des conditionnelles sont courtes. En effet, une instruction de modification du masque peut provoquer l'arrêt du séquencement tant que la nouvelle valeur du masque n'est pas présente sur l'entrée du séquenceur, mais elle ne provoque pas de mise à jour du compteur ordinal. Cependant, pour permettre le choix programmé de la longueur maximum des portions de vecteurs traités, il faut,

soit prévoir la mémorisation et les instructions d'initialisation de cette valeur dans le séquenceur, soit traduire les boucles vectorielles par des boucles séquentielles et gérer la longueur des vecteurs explicitement dans ces boucles comme on le fait pour le CRAY 1.

11 les instructions de réduction

Le traitement des opérations de réduction est souvent délicat sur les architectures pipelines. Dans le cas du DSPA, le résultat d'une instruction d'un opérateur peut être émis vers une entrée logique de ce même opérateur. L'exécution de la boucle de la figure 27 provoque le cumul

Sq	initBS, Ri	
M	lpic,Da	+fd
+f	+, +f, M	+fg
Sq	itère, Ri	

figure 27

des données lues en mémoire sur la FIFO associée au point de connexion sortie additionneur flottant → entrée logique gauche de l'additionneur flottant. Mais, tout dépend du nombre d'éléments initiaux de cette FIFO. Si avant la première itération, cette FIFO contient un zéro, elle contient la somme des valeurs lues en mémoire à la fin de la dernière itération. Mais, le pipeline de l'additionneur ne peut alors contenir qu'un seul traitement à un instant donné. Par contre, supposons que cette FIFO contienne 4 zéros avant de rentrer dans la boucle. Dans ce cas, à la fin de la boucle, cette FIFO contient 4 sous-sommes et le parallélisme potentiel du pipeline de l'additionneur est alors de 4. Cette instruction peut également être exécutée en mode vectoriel, le nombre de sous sommes obtenus étant égal au nombre de valeurs initiales (et non à la longueur des vecteurs qui peut d'ailleurs être inférieur à 4). Dans le cas où le masquage est traité par compression, cette instruction est toujours applicable, la FIFO contenant alors 4 sous sommes des éléments non masqués. Pour obtenir un parallélisme maximum dans le calcul de ces sous sommes, il suffit que le nombre de sous sommes soit supérieur ou égal au nombre

d'étages du pipeline de l'unité fonctionnelle. La seule difficulté de ce rebouclage direct provient du fait qu'il faut placer les valeurs initiales dans la bonne FIFO, et qu'il faut sommer les valeurs finales. Les valeurs initiales ne peuvent être placées que par l'unité fonctionnelle elle-même (seule productrice dans cette FIFO). Il faut, par exemple, émettre les valeurs initiales vers l'unité fonctionnelle, et coder autant d'instructions identité sur l'unité fonctionnelle (figure 28). Cette

```
Ml |   lire,zero    | +fg
Ml |   lire,zero    | +fg
Ml |   lire,zero    | +fg
Ml |   lire,zero    | +fg
+f |   identite,Ml  | +fg
+f |   identite,Ml  | +fg
+f |   identite,Ml  | +fg
+f |   identite,Ml  | +fg
```

figure 28

instruction identité se contente d'émettre la donnée reçue sur l'entrée logique gauche vers le destinataire désigné. Une autre solution consiste à prévoir sur l'additionneur des instructions d'initialisation. Dans tous les cas, cette unité fonctionnelle doit exécuter autant d'instructions d'initialisation qu'il y a de sous sommes, ce nombre pouvant être codé dans l'instruction.

```
+f |   initsomme,4 | +fg
```

Cette instruction émet 4 zéros vers le destinataire désigné.

Pour le calcul de la somme finale, le problème provient du fait que toutes les sous sommes sont sur la même entrée logique. Là aussi, une première solution consiste à transférer la moitié des sous sommes sur l'autre entrée logique par des instructions identité (figure 29). Une autre solution (figure 30) consiste à définir une instruction d'addition qui prend ses deux opérandes sur l'entrée logique gauche, · instruction exécutable uniquement en scalaire. Ces opérations de réduction peuvent

+f	identite,+f	+fd
+f	identite,+f	+fd
+f	+, +f, +f	+fg
+f	+, +f, +f	+fd
+f	+, +f, +f	destinataire de la somme

figure 29

+f	+S,	+fg
+f	+S,	+fd
+f	+, +f, +f	destinataire de la somme

figure 30

être définies pour de nombreuses opérations (Somme, Produit, etc). Leur utilisation n'est pas limitée aux réductions sur des vecteurs lus en mémoire, mais possible sur toute suite de données internes au processeur. Par exemple, le produit scalaire de deux vecteurs peut être codé par la boucle vectorielle de la figure 31 précédé par l'une des séquences d'initialisation proposées, et suivie par l'une des séquences de sommation finale. La seule condition à respecter étant que lorsque la FIFO de

Sq	initBV, M1	
M	Vlpic,Dx	*fg
M	Vlpic,Dy	*fd
*f	VV*, M, M	+fd
+f	VV+, +f, M	+fg
Sq	itèreVL	

figure 31

rebouclage est utilisée dans le calcul de l'expression vectorielle à sommer, le rebouclage direct soit remplacé par un rebouclage à travers une

autre unité fonctionnelle (mémoire locale, par exemple), cas très particulier et très rare. Ce même alogrithme peut également être appliqué même lorsqu'il y a masquage des vecteurs dans l'itération vectorielle à condition que le masquage soit traité par compression sur les FIFOs. La génération du code correspondant à ces réductions ne pose aucun problème spécifique de compilation. En effet, tous les opérateurs de réduction sont prédéfinis dans Hellena, ce qui élimine les phénomènes de rebouclage sur les arbres d'expressions.

12 les exeptions et les interruptions

Dans les processeurs d'usage général, le séquencement d'un programme peut être interrompu de manière non programmée, soit pour prendre en compte des évènements externes au programme, soit pour prendre en compte des évènements déclenchés par le programme lui-même. Ces ruptures de séquence entraînent l'exécution de séquences de traitement et permettent le partage des ressources de calcul d'un processeur entre plusieurs programmes (multiprogrammation) ou le lancement de routines du système. Pour que la commutation d'un programme à un autre soit possible, l'état courant d'un processus doit pouvoir être matérialisé. Suspendre l'exécution d'un programme consiste alors à sauvegarder son état en mémoire. L'exécution du programme peut être reprise en restaurant cet état. Sur une machine séquentielle, l'état d'un programme peut être matérialisé par la valeur du compteur ordinal, l'ensemble des registres ainsi qu'un mot d'état du processeur. Sur une machine parallèle, l'ensemble des informations qui constituent l'état courant d'un programme peut être très large. Dans le cas général, on peut considérer qu'il y a d'autant plus d'informations dans cet état qu'il y a de parallélisme dans l'architecture. Les opérations de suspension et de reprise sont donc d'autant plus coûteuses qu'il y a de parallélisme.

On peut limiter l'ensemble des informations d'état d'un processus de deux manières. La première consiste à n'interrompre un processus que par des processus qui n'utilisent qu'une faible partie des ressources du processeur et donc de réduire la quantité d'information d'état des processus. Cette solution permet de réaliser des tâches de gestion (entrées-sorties, etc), mais interdit une véritable multiprogrammation.

Elle est acceptable sur le domaine d'application des calculateurs qui nous intéressent. Sur le Cray 1, par exemple, les registres vectoriels ne sont pas sauvegardés. La deuxième solution consiste à n'interrompre un programme qu'en des points prévus. Il suffit alors, soit de restreindre l'état d'un processus à un minimum d'informations dans ces points, soit d'associer à ces points des routines de sauvegarde et de restauration explicites. C'est ce qui se passe en général dans les programmes multitâches avec synchronisations explicites.

Dans le cas du DSPA, l'état d'un processus à un instant donné est composé des états de chaque unité fonctionnelle et du contenu de chaque FIFO. Sauvegarder cet état reviendrait à sauvegarder l'état de chaque unité fonctionnelle et chaque FIFO, opération irréalisable dans la pratique. Il est possible de définir un sous état qui permette d'exécuter des routines de gestion système, mais ces routines systèmes ne pourraient pas utiliser les FIFOs d'échange du programme interrompu. Nous pensons donc qu'il est plus simple d'associer au processeur DSPA un deuxième processeur de gestion de la machine et de prévoir un système d'échange d'informations entre le DSPA et ce processeur. C'est la solution classique où un coprocesseur de calcul rapide est associé à un système d'usage général. Pour pouvoir exécuter des algorithmes multitâches sur un DSPA, il faut cependant être capable d'interrompre les programmes en des points prédéterminés des programmes (par exemple, aux points de synchronisation explicites). La solution la plus simple consiste alors à produire le code de telle façon que, si le séquencement d'un programme est arrêté sur une telle instruction, le séquencement de chaque unité fonctionnelle est tel que toutes les instructions déjà séquencées seront exécutées. Il suffit alors d'attendre que toutes les unités fonctionnelles aient vidé leur FIFO d'instructions pour limiter l'état du DSPA au contenu des registres internes (descripteurs internes au séquenceur, à certaines unités fonctionnelles et mémoire locale). Ces instructions interruptibles peuvent être indiquées, par exemple, par un bit de chaque mot de la mémoire de programme. Notons enfin que l'attente de vidage des pipelines n'est pas strictement nécessaire. Il suffirait que le séquenceur place dans la FIFO d'instructions de chaque unité fonctionnelle une instruction de sauvegarde des descripteurs internes (émission des descripteurs vers la mémoire, par exemple) suivie d'une instruction de restauration des descripteurs (pour le nouveau processus) pour que le vidage complet ne se produise pas. Ces

problèmes d'interruption sont repris en [Jégou ?] dans la définition du processeurs SDSPA pour lequel l'état d'un programme peut être matérialisé en permanence.

Les interruptions externes ne sont pas les seuls événements à traiter sur un processeur. Il faut également être capable de traiter tous les problèmes d'erreurs produites par l'exécution des instructions du programme. Comme il n'y a pas de relation directe entre les séquencements internes des unités fonctionnelles, ces événements ne peuvent pas être traités par déroutement du séquenceur. La solution proposée consiste à vérifier quelles erreurs ont eu lieu entre deux points du programme. Pour celà, à chaque donnée interne d'un processeur est associé un historique d'erreur, par exemple, un bit pour les *overflows*, un bit pour les *underflows*, etc. Ces bits sont mis à jour par les opérateurs. Par exemple, si un overflow se produit sur une instruction de l'additionneur flottant, cette erreur est mémorisée dans le bit correspondant du résultat de l'opération. De plus, l'historique d'erreur de tout résultat d'opération contient l'union des historiques des opérandes de l'instruction qui produit ce résultat. Certaines unités fonctionnelles absorbent les données du processeur. C'est le cas de la mémoire pour les données à écrire et leurs adresses, de la mémoire locale, et du séquenceur pour les conditions (instructions qui ne produisent pas de résultat). Nous associons à chacune de ces unités fonctionnelle un historique d'erreurs des données absorbées. C'est l'union des historiques d'erreurs. Il suffit alors de prévoir sur ces unités fonctionnelles une instruction d'initialisation de l'historique, et une instruction de lecture de cet historique. Une instruction de lecture d'historique émet généralement la valeur courante de l'historique sur une entrée logique du séquenceur. Pour savoir quelles erreurs se produisent entre deux points d'un programme (par exemple pendant l'exécution d'une procédure), il suffit d'émettre vers les unités fonctionnelles concernées une instruction d'initialisation d'historique d'erreurs au début de la séquence de code à contrôler, d'émettre vers ces unités fonctionnelles les instructions de lecture de ces historiques à la fin du code, et de coder une instruction de saut conditionnelle aux valeurs reçues sur le séquenceur.

Cette technique ne permet pas de savoir à quel point (valeur correspondante du compteur ordinal) du programme des erreurs sont produite.

Elle est cependant conforme aux systèmes de traitement des exceptions du langage Ada (traitement des erreurs d'un bloc sans possibilité de correction). Pour effectuer des mises au point de programme, ces test peuvent affinés, par exemple pour chaque itération d'une boucle, ou même sur une instruction. Mais un traitement trop fin a pour effet de réduire le parallélisme des pipelines car le saut conditionnel n'est exécuté qu'après réception des historiques, et donc qu'après que toutes les données à écrire en mémoire avant la lecture de l'historique aient été effectivement écrites. Cette modification du parallélisme entraîne de plus que les erreurs dûes à l'asynchronisme d'accès en mode MIMD à la mémoire peuvent changer.

13 considérations de mise en oeuvre

Les unités fonctionnelles d'un DSPA sont indépendantes. Une mise en oeuvre de ce processeur dépend donc fortement des choix d'implantation des FIFOs. Les FIFOs d'un DSPA sont regroupées aux points de connexion d'un réseau crossbar. Pendant chaque cycle, une seule des FIFOs en sortie d'une même unité fonctionnelle peut recevoir une donnée. De même, à chaque cycle, une seule des FIFOs associées à la même entrée logique d'une même unité fonctionnelle peut émettre une donnée. Toutes les unités fonctionnelles d'un DSPA ne produisent pas une donnée à chaque cycle, ni ne consomment pas une donnée à chaque cycle sur chaque entrée logique. Par exemple, la mémoire peut produire une donnée à chaque cycle alors que les opérateurs flottant n'exécutent qu'une instruction tous les deux cycles. De plus, les unités fonctionnelles sont asynchrones. Elles peuvent se partager certaines ressources physiques du processeur. Toutes ces considérations permettent de définir certaines possibilités de mise en oeuvre de ces FIFOs.

La mise en oeuvre la plus évidente consiste à intégrer le réseau de FIFO sur un seul circuit. Ce réseau possède n entrées et m sorties, n étant le nombre de sorties d'unités fonctionnelles et m le nombre total d'entrées logiques du processeur. Mais, le nombre de *pattes* des *puces* entraînerait une limitation de la largeur des FIFOs à quelques bits. De plus, pour simplifier le séquencement des unités fonctionnelles, il faut que les signaux *FIFO pleine* et *FIFO vide* soient présents en permanence. Ce

nombre de pattes peut être diminué en considérant les unités fonctionnelles à faible débit. Par exemple, il est possible de partager statiquement un bus de sortie entre deux unités fonctionelles qui ne produisent qu'un seul résultat un cycle sur deux (en allouant le cycle pair à l'une des unités fonctionnelles, et le cycle impair à l'autre unité). Il suffit alors d'ajouter un bit (une parité d'horloge) pour l'adressage des FIFOs connectées à ce bus. Certaines unitées fonctionnelles émettent rarement des données. Dans ce cas, un bus de données propre peut être relativement coûteux. Le DSPA n'interdit pas le partage dynamique de ressources internes. Ce partage doit permettre le séquencement simple d'opérateurs pipeline. Il faut en général éviter que la phase d'émission de résultats échoue pour cause d'occupation d'une ressource. Ces conflits sont résolus simplement par la mise en place de systèmes de réservation. Un tel système de réservation peut être constitué d'un registre à décalage V_0, ..., V_n. Au cycle t, la valeur V_i indique l'état de la réservation pour le cycle $t+i$. La valeur V_0 indique l'état de la réservation pour le cycle courant. Cet état peut contenir certaines informations comme l'identité de l'unité fonctionnelle propriétaire de cette réservation. Chaque unité fonctionnelle émet ses requêtes de réservations un certain nombre de cycles d avant l'émission du résultat correspondant. La réservation est acceptée si la valeur V_d correspondante indique que le cycle $t+d$ est libre. Cette acceptation entraîne l'occupation du cycle. Le rejet de la réservation entraîne l'arrêt du séquencement de l'unité fonctionnelle, la demande de réservation étant généralement réémise au cycle suivant. La mise en oeuvre d'un tel système est très simple lorsque toutes les unités fonctionnelles qui se partagent une certaine ressource émettent leurs demandes de réservation avec des délais d différents. Dans ce cas, l'unité fonctionnelle qui possède le délai le plus long, et donc le plus grand nombre d'étages de pipeline devient prioritaire. Lorsque des délais sont égaux, un arbitrage doit être intégré au système. L'utilisation d'un tel système peut être également généralisé à la réservation d'emplacements dans les FIFOs. En effet, la progression des opérations dans les opérateurs pipeline ne peut pas toujours être suspendue. Dans ce cas, il faut s'assurer que, lorsqu'une donnée est produite par un opérateur, elle peut toujours être émise vers le destinataire. Il faut donc s'assurer de la disponibilité des ressources requises pour l'émission d'un résultat avant d'initialiser une opération pipeline. Parmis ces ressources, certaines sont partagées, la réservation pouvant alors se faire par le système

proposé précédemment, d'autres sont propres à l'unité fonctionnelle comme les FIFOs de sortie. Il faut s'assurer à l'initialisation de chaque opération que l'écriture du résultat dans la FIFO destinatrice sera possible *d* cycles plus tard. Deux solutions simples existent. La première consiste à lever le signal *FIFO pleine* dès qu'un certain nombre de données sont mémorisées dans une FIFO de capacité supérieure. Les éléments supplémentaires des FIFO permettent alors de garantir que les résultats de toutes les opérations en cours dans le pipeline peuvent être mémorisées. Une autre solution consiste à associer un système de réservation à chaque FIFO. Ce système peut être réduit à un compteur/décompteur incrémenté de un à chaque fois qu'une réservation est acceptée et décrémenté de un à chaque fois qu'une donnée est extraite de la FIFO. Les réservations sont refusées lorsque la valeur de ce compteur est égale à la capacité de la FIFO. Comme une réservation a été demandé préalablement à l'émission de tout résultat, on garantit que chaque résultat peut être mémorisé.

En prenant tous ces paramètres en considération, on peut définir plusieurs implantations possibles du DSPA.

↦ cablage du réseau de FIFO.

↦ réalisation de plus petits réseaux (par exemple, un réseau 4x4). Plusieurs petits réseaux peuvent être concaténés pour former un plus grand réseau. Cette solution permet de tirer partie du fait que tous les points de connexion du réseau ne sont pas utilisés. Pour tirer partie des différences de débit des unités fonctionnelles, il est possible de réaliser des réseaux plus grand (8x8 par exemple) dont certains bus internes se partagent des ports de lecture et d'écriture. On peut également gérer à ce niveau les données en simple et en double précision. Un réseau 8x8 peut, par exemple, être utilisé comme un réseau 4x4 en double précision, ce qui permet des échanges de doubles mots au débit nominal. Lorsque le débit en double précision est deux fois moindre en lecture et en écriture, les deux demi-mots peuvent être rangés consécutivement dans des FIFO de demi-mots. Mais il est également possible de s'adapter à des débits différents en lecture et en écriture en utilisant deux FIFOs de demi-mots en parallèle. Les accès au débit le plus faible se font en alternant les accès aux deux FIFOs. La prise en compte de ces diverses possibilités doit permettre de

définir un circuit unique utilisable dans plusieurs modes de fonctionnement.

↦ réalisation du réseau à partir de mémoires rapides et de circuits de gestion de FIFO qui émettent des adresses vers ces mémoires. Cette solution permet de n'utiliser que des circuits existants (mémoires et PLA), mais ne permet pas une intégration aussi compacte. Chacune de ces mémoire peut supporter plusieurs FIFOs. Ce repliage dépend du débit des mémoires utilisées (nombre d'écritures et de lectures par cycle), du débit des unités fonctionnelles qui peuvent écrire, et du débit des unités fonctionnelles qui peuvent lire sur une même mémoire. Par exemple, en utilisant des fichiers de registres qui permettent trois écritures et deux lectures par cycle, il est possible de rassembler toutes les FIFO correspondant aux entrées logiques de deux opérateurs flottants dans un seul bloc de registres de la manière suivante :

- Un port d'écriture est associé au bus de sortie de la mémoire, ce qui permet l'emission d'une donnée lue en mémoire vers les opérateurs flottants à chaque cycle.
- Chaque opérateur flottant exécute une instructions tous les deux cycles, et émet donc une donnée tous les deux cycles. En décalant leurs émissions d'un cycle, ces deux opérateurs se partagent un bus de sortie statiquemement, et se partagent donc un port d'écriture dans le fichier de registres. Le troisième port est partagé dynamiquement entre les autres unités fonctionnelles (par exemple à travers un système de réservation). Notons à ce sujet qu'une unité fonctionnelle peut émettre sur plusieurs bus de sortie sans difficulté.
- Les deux entrées logiques gauche et droite de chaque opérateur sont repliés sur les ports de lecture du fichier de registres (lecture d'une donnée sur chaque entrée logique tous les deux cycles).

Chaque point de connexion est contrôlé par un gestionnaire de FIFO qui émet des adresses vers les entrées d'adresses associées aux ports de lecture et d'écriture. De tels repliages des FIFOs dans des registres sont possibles sur l'ensemble des unités fonctionnelles du DSPA.

↦ les bus de transport des données peuvent être associés aux sorties et aux entrées des unités fonctionnelles à haut débit, partagés statiquement entre des unités fonctionnelles à débit plus faible (par exemple un cycle sur deux sur un bus partagé entre deux unités fonctionnelles), partagés

dynamiquement (avec arbitrage) entre des unités fonctionnelles peu requises.

14 conclusion

L'objectif fixé qui était d'exécuter rapidement du code séquentiel irrégulier a abouti à la définition d'un processeur pipeline synchronisé par les données, le DSPA. Cet objectif a été atteint en limitant au minimum les opérations à dérouler pendant la phase de séquencement des instructions. Chaque unité fonctionnelle possède son propre séquenceur et son propre décodeur spécialisé d'instructions. Cette stratégie a permis de localiser la gestion du comportement de chaque unité fonctionnelle dans son propre séquenceur d'instructions. L'utilisation optimale d'opérateurs pipelines et de mémoires entrelacées est simplifiée sur le DSPA. De plus, la production de codes rapides pour ce processeur reste très simple, ce qui évite l'emploi de logiciels d'optimisation complexes. L'indépendance des unités fonctionnelles ainsi que l'absence de tout système global de gestion des données internes laissent espérer une mise en oeuvre efficace et peu coûteuse d'un tel processeur. Nous étudions en [Jégou 86a] les possibilités offertes lorsqu'un tel processeur est intégré dans une architecture multiprocesseur.

Références

[Jégou 86a]

Jégou Y., *Le multi DSPA, un multiprocesseur pipeline synchronisé par les données*, à paraître dans les publications internes de l'IRISA.

[Jégou 86b]

Jégou Y., *Hellena, un langage pour les calculateurs vectoriels*, à paraître dans les publications internes de l'IRISA.

[Jégou 86c]

Jégou Y., *Hellena, principes de mise en oeuvre*, à paraître dans les publications internes de l'IRISA.

[Jégou ?]

Jégou Y., *Le SDSPA, un processeur pipeline universel*, à paraître dans les publications internes de l'IRISA.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique